SISTEMAS DISTRIBUIDOS

Principios y Paradigmas



Segunda edición



Andrew S. Tanenbaum
Maarten Van Steen

SISTEMAS DISTRIBUIDOS

Principios y Paradigmas

Segunda edición

Acerca de los autores

Andrew S. Tanenbaum es egresado del MIT de la licenciatura en Física, y cuenta con un doctorado en investigación de la University of California, en Berkeley. En la actualidad es profesor de Ciencias de la Computación en Vrije Universiteit en Amsterdam, Holanda, donde encabeza al Grupo de Sistemas de Cómputo. Durante 12 años, hasta que renunció en enero de 2005, se desempeñó como Decano de la Escuela Superior de Computación e Imagen, una institución de estudios de posgrado que investiga sistemas paralelos avanzados, distribuidos y de imágenes.

En el pasado, realizó investigaciones sobre compiladores, sistemas operativos, redes y sistemas distribuidos de área local. Su investigación actual se enfoca principalmente en la seguridad en las computadoras, en especial de sistemas operativos, redes y de sistemas distribuidos de área amplia. Juntos, todos estos proyectos de investigación han derivado en más de 125 artículos y ponencias evaluadas por expertos y cinco libros, los cuales se han traducido a 21 idiomas.

El profesor Tanenbaum también ha producido una cantidad considerable de software. Fue el principal arquitecto del Amsterdam Compiler Kit, un juego de herramientas para escribir compiladores portables, así como de MINIX, un pequeño clon de UNIX que tiene un alto grado de confiabilidad, el cual está disponible de manera gratuita en *www.minix3.org*. Este sistema fue la inspiración y la base en la que se desarrolló Linux. También fue también uno de los diseñadores en jefe de Amoeba y Globe.

Sus estudiantes de doctorado se llenan de gloria después de conseguir sus títulos. Él está muy orgulloso de ellos. En este sentido, parece mamá gallina.

El profesor Tanenbaum es socio de la ACM, del IEEE y miembro de la Royal Netherlands Academy of Arts and Sciences. Además, en 1994 ganó el premio Karl V. Karlstrom Outstanding Educator de la ACM, en 1997 obtuvo el premio ACM/SIGCSE por sus destacadas contribuciones a la educación en la ciencia de la computación y en 2002 ganó el premio Texty por su excelencia en libros de texto. En 2004, la Royal Academy lo nombró uno de los cinco nuevos Academy Professors. Su página personal se encuentra en *www.cs.vu.nl/~ast*.

Maarten Van Steen es profesor en la Vrije Universiteit, Amsterdam, donde enseña sobre sistemas operativos, redes de computadoras y sistemas distribuidos. También ha impartido diversos cursos muy exitosos acerca de temas relacionados con sistemas de cómputo para profesionales de la industria y organizaciones gubernamentales.

El profesor Van Steen estudió matemáticas aplicadas en Twente University y obtuvo el grado de Doctor en Ciencias de la Computación de la Leiden University. Después de sus estudios universitarios, trabajó para un laboratorio de investigación industrial, en el que se desempeñó como líder del grupo de sistemas de cómputo y se concentró en el soporte para la programación de aplicaciones paralelas.

Después de cinco años de esforzarse por investigar y administrar simultáneamente, decidió retomar la academia, primero como profesor asistente de ciencias de la computación en la Erasmus University, de Rotterdam, y después como profesor asistente en el grupo de Andrew Tanenbaum, en la Vrije Universiteit, Amsterdam. Volver a la universidad fue una decisión acertada; su esposa también piensa lo mismo.

Su investigación actual se concentra en sistemas distribuidos de gran escala. Parte de su investigación se enfoca en los sistemas basados en la web, en particular sobre distribución adaptativa y replicación en Globule, una red de entrega de contenidos, de la que su par Guillaume Pierre es diseñador en jefe. Otro tema de amplia investigación es el de sistemas de igual a igual totalmente descentralizados (basados en el gossip), cuyos resultados se han incluido en Tribler, una aplicación de BitTorrent desarrollada en colaboración con académicos de la Technical University de Delft.

SISTEMAS DISTRIBUIDOS

Principios y Paradigmas

Segunda edición

Andrew S. Tanenbaum
Maarten Van Steen

Vrije Universiteit
Amsterdam, Holanda

TRADUCCIÓN

Jorge Octavio García Pérez
Ingeniero en Computación
Universidad Nacional Autónoma de México

Rodolfo Navarro Salas
Ingeniero Mecánico
Universidad Nacional Autónoma de México

REVISIÓN TÉCNICA

Aarón Jiménez Govea
Catedrático del Departamento de Ciencias Computacionales
Universidad de Guadalajara, México



México • Argentina • Brasil • Colombia • Costa Rica • Chile • Ecuador
España • Guatemala • Panamá • Perú • Puerto Rico • Uruguay • Venezuela

Datos de catalogación bibliográfica

TANENBAUM, ANDREW S. y
MAARTEN VAN STEEN

Sistemas Distribuidos. Principios y Paradigmas
Segunda edición

PEARSON EDUCACIÓN, México, 2008

ISBN: 978-970-26-1280-3
Área: Computación

Formato: 18.5 × 23.5 cm Páginas: 704

ISBN 9780132392273

Edición en español

Editor: Luis Miguel Cruz Castillo
e-mail:luis.cruz@pearsoned.com
Editor de desarrollo: Bernardino Gutiérrez Hernández
Supervisor de producción: Gustavo Rivas Romero

Edición en inglés

Vice President and Editorial Director, ECS: Marcia J. Horton
Executive Editor: Tracy Dunkelberger
Editorial Assistant: Christianna Lee
Associtate Editor: Carole Snyder
Executive Managing Editor: Vince O'Brien
Managing Editor: Camille Trentacoste
Production Editor: Craig Little
Director of Creative Services: Paul Belfanti
Creative Director: Juan Lopez
Art Director: Heather Scott
Cover Designer: Tamara Newnam
Art Editor: Xiaohong Zhu
Manufacturing Manager, ESM: Alexis Heydt-Long
Manufacturing Buyer: Lisa McDowell
Executive Marketing Manager: Robin O'Brien
Marketing Assistant: Mack Patterson

SEGUNDA EDICIÓN, 2008

D.R. © 2008 por Pearson Educación de México, S.A. de C.V.
Atlacomulco 500-5o. piso
Col. Industrial Atoto
53519, Naucalpan de Juárez, Estado de México

Cámara Nacional de la Industria Editorial Mexicana. Reg. Núm. 1031.



ISBN 10: 970-26-1280-2

ISBN 13: 978-970-26-1280-3

Impreso en México. *Printed in Mexico*.

1 2 3 4 5 6 7 8 9 0 11 10 09 08

Para Suzanne, Barbara, Marvin y en memoria de Bram y Sweetie π

AST

Para Mariëlle, Max y Elke

MVS

CONTENIDO

PREFACIO

Los sistemas distribuidos forman parte de un campo muy cambiante de las ciencias de la computación. Desde la primera edición de este libro, han emergido nuevos y estimulantes temas tales como la computación punto a punto (*peer-to-peer*) y redes de monitoreo, mientras que otros se han vuelto más maduros, como los servicios web y las aplicaciones web en general. Cambios como los anteriores requirieron que revisáramos nuestro texto original para actualizarlo.

Esta segunda edición refleja una importante revisión comparada con la anterior. Agregamos un capítulo independiente sobre arquitecturas para reflejar el progreso que se ha dado en la organización de los sistemas distribuidos. Otra diferencia importante es que ahora existe mucho más material acerca de sistemas descentralizados, en especial sobre el cómputo punto a punto. No solamente explicamos las técnicas básicas, también ponemos atención en sus aplicaciones, tales como archivos compartidos, la diseminación de información, las redes de contenido distribuido, y los sistemas de publicación-suscripción.

Después de estos dos importantes temas, explicamos otros nuevos a lo largo del libro. Por ejemplo, incluimos material relacionado con redes de monitoreo, virtualización, servidores de clusteres (*Server clusters*) y cómputo con base en mallas de computadoras (*grid computing*). Ponemos especial atención en la autoadministración de sistemas distribuidos, un tema que cada vez adquiere mayor relevancia mientras los sistemas aumentan en escala.

Por supuesto, también modernizamos todo material que consideramos pertinente. Por ejemplo, cuando explicamos la consistencia y la replicación, ahora nos enfocamos en los modelos consistentes que son más apropiados para los sistemas distribuidos modernos en lugar de los modelos originales, los cuales adaptamos para sistemas distribuidos de alto rendimiento. Asimismo, agregamos

material relacionado con algoritmos distribuidos modernos, incluidos los algoritmos de sincronización de reloj basados en GPS y algoritmos de localización.

Aunque es inusual, hemos sido capaces de *reducir* el número total de páginas. En parte, esta reducción es causada por la eliminación de temas tales como el recolector distribuido de basura y los protocolos de pago electrónico, y también por la reorganización de los cuatro primeros capítulos.

Como en la edición anterior, el libro está dividido en dos partes. Principios de los sistemas distribuidos en los capítulos del 2 al 9, y los métodos con respecto a la forma en cómo se deben desarrollar los sistemas distribuidos (los paradigmas) en los capítulos del 10 al 13. Sin embargo, a diferencia de la edición previa, decidimos no explicar casos de estudio completos dentro de los capítulos de paradigmas. Al contrario, relacionamos cada principio a través de un caso representativo. Por ejemplo, explicamos las invocaciones de objetos como un principio de comunicación dentro del capítulo 10 que trata sobre sistemas distribuidos basados en objetos. Este método nos permite condensar el material, pero también hacerlo más fácil y atractivo de leer.

Por supuesto, continuaremos basándonos en la práctica para explicar de qué tratan los sistemas distribuidos. Explicaremos distintos aspectos de los sistemas aplicados a la vida real, tales como Web-Sphere MQ, DNS, GPS, Apache, CORBA, Ice, NFS, Akamai, TIB/Rendezvous, Jini y muchos otros. Estos ejemplos explican la delgada línea presente entre la teoría y la práctica, y hacen de los sistemas distribuidos un campo excitante.

Muchas personas han contribuido de distinta manera en la elaboración del libro. Queremos agradecer de modo especial a D. Robert Adams, Arno Bakker, Coskun Bayrak, Jacques Chassin de Kergommeaux, Randy Chow, Michel Chaudron, Punnet Singh Chawla, Fabio Costa, Cong Du, Dick Epema, Kevin Fenwick, Chandana Gamage, Ali Ghodsi, Giorgio Ingargiola, Mark Jelasity, Ahmed Kamel, Gregory Kapfhammer, Jeroen Ketema, Onno Kubbe, Patricia Lago, Steve MacDonald, Michael J. McCarthy, M. Tamer Ozsu, Guillaume Pierre, Avi Shahar, Swaminathan Sivasubramanian, Chintan Shah, Ruud Stegers, Paul Tymann, Craig E. Wills, Reuven Yagel y Dakai Zhu, por leer partes del manuscrito, identificar los errores en la edición anterior y ofrecer comentarios útiles.

Por último, queremos agradecer a nuestras familias. Suzanne ya ha estado en este proceso durante diecisiete veces. Éstas son muchas veces para mí, pero también para ella. Ninguna vez ha dicho: “es suficiente”. No obstante, seguramente este pensamiento ha rondado por su cabeza. Gracias. Ahora Barbara y Marvin tienen una mejor idea de lo que los profesores hacen para vivir y conocen la diferencia entre un buen libro de texto y uno malo. Ellos son ahora una inspiración para mí para intentar producir más libros buenos que malos (AST).

Debido a que me tomé un año sabático para actualizar el libro, todo el negocio de escribir fue más atractivo para Mariëlle. Comienza a acostumbrarse, me apoya, pero me alerta cuando es tiempo de redirigir la atención hacia temas más importantes. Le debo mucho. Por ahora, Max y Elke tienen una mejor idea de lo que significa escribir un libro, pero comparado con lo que ellos mismos leen, les cuesta trabajo comprender dónde reside lo excitante acerca de estas cosas extrañas llamadas sistemas distribuidos. No los puedo culpar (MVS).

1

INTRODUCCIÓN

Los sistemas computacionales están experimentando una revolución. De 1945, cuando comenzó la era moderna de las computadoras, a 1985, éstas eran grandes y caras. Incluso las minicomputadoras costaban al menos decenas de miles de dólares. Como resultado, muchas empresas tenían solamente unas cuantas, y debido a la falta de un medio de conexión entre ellas, operaban de manera independiente.

Sin embargo, hacia la mitad de la década de 1980, dos avances en la tecnología comenzaron a cambiar esa situación. El primero de estos avances fue el desarrollo de poderosos microprocesadores. Inicialmente, los microprocesadores eran máquinas de 8 bits, pero pronto se hicieron comunes las CPU de 16, 32 y 64 bits. Muchas de ellas tenían el poder de una mainframe (es decir, una computadora grande), pero a una fracción de su precio.

La cantidad de mejoras que han tenido lugar en la tecnología de las computadoras a partir de la segunda mitad del siglo XX es verdaderamente impresionante, y no tiene precedente en otras industrias. De una máquina que costaba 10 millones de dólares y ejecutaba 1 instrucción por segundo, saltamos a máquinas que cuestan 1000 dólares y son capaces de ejecutar un millón de millones de instrucciones por segundo; esto significa una ganancia precio/rendimiento de 10^{13}. Si los automóviles hubieran mejorado a ese grado en el mismo periodo, en la actualidad un Rolls Royce costaría 1 dólar y tendría un rendimiento de 1 millón de millones de millas por galón de combustible. (Por desgracia, tendría un manual de 200 páginas para indicarle cómo abrir la puerta.)

El segundo desarrollo importante fue la invención de las redes de computadoras de alta velocidad. Las **redes de área local,** o **LAN** (*local-area networks*), permiten la interconexión de cientos de máquinas localizadas dentro de un mismo edificio, de tal manera que es posible transferir

pequeños volúmenes de información entre máquinas en unos cuantos microsegundos, más o menos. Podemos transferir grandes volúmenes de datos entre máquinas a velocidades que van de los 100 millones a los 10 millones de millones de bits/segundo. Las **redes de área amplia**, o **WAN** (*wide-area networks*), permiten la interconexión de millones de máquinas ubicadas alrededor del mundo a velocidades que van desde los 64 Kbps (kilobits por segundo) hasta gigabits por segundo.

El resultado de estas tecnologías es que ahora no solamente es factible, sino fácil, poner a trabajar sistemas de cómputo compuestos por grandes cantidades de computadoras interconectadas mediante una red de alta velocidad. Por lo general, a estos sistemas se les conoce como redes de computadoras o **sistemas distribuidos**, al contrario de los **sistemas centralizados** (o **sistemas de un solo procesador**) que por lo general constan de una sola computadora, sus periféricos, y quizás algunas terminales remotas.

1.1 DEFINICIÓN DE UN SISTEMA DISTRIBUIDO

En la literatura especializada, existen distintas definiciones para los sistemas distribuidos, ninguna de ellas satisfactoria, y ninguna guarda concordancia con las demás. Para nuestro propósito, es suficiente la siguiente definición breve:

> *Un sistema distribuido es una colección de computadoras independientes que dan al usuario la impresión de constituir un único sistema coherente.*

Esta definición comprende diversos aspectos importantes. En primer lugar, tenemos que un sistema distribuido consta de componentes (es decir, computadoras) autónomos. El segundo aspecto es que los usuarios (personas o programas) creen que realmente interactúan con un sistema único. Esto significa que de una manera o de otra los componentes autónomos necesitan colaborar entre sí. La forma de establecer la colaboración radica en el fondo del desarrollo de los sistemas distribuidos. Observe que no hacemos suposiciones con respecto al tipo de computadoras. En principio, incluso con un sistema individual, éstas podrían componerse de computadoras *mainframe* de alto rendimiento o de pequeños nodos ubicados dentro de redes de sensores. De manera similar, no hacemos suposiciones acerca de la manera en que se interconectan las computadoras. Más adelante volveremos a tratar estos aspectos.

En lugar de ir más allá con definiciones, quizá resulte más útil concentrarnos en las características de los sistemas distribuidos. Una característica importante es que las diferencias entre las distintas computadoras y la manera en que se comunican entre sí quedan ocultas para el usuario. Lo mismo sucede con la organización interna de un sistema distribuido. Otra característica importante es que los usuarios y las aplicaciones pueden interactuar con un sistema distribuido de manera consistente y uniforme, sin importar dónde y cuándo tenga lugar.

En principio, los sistemas distribuidos también debieran ser fáciles de expandir o escalar. Esta característica es consecuencia directa de tener computadoras independientes, pero al mismo tiempo, de ocultar cómo estas computadoras realmente forman parte del sistema como un todo. Por lo general, un sistema distribuido estará disponible de manera continua, aunque tal vez algunas partes pudieran encontrarse fuera de operación. Los usuarios y las aplicaciones no deben notar que las partes son reparadas, o que se agregan nuevas secciones para servir a más usuarios o aplicaciones.

Con el objeto de dar soporte a computadoras y redes heterogéneas mientras se ofrece la vista de un sistema único, los sistemas distribuidos se organizan a menudo en términos de una capa de software, esto es, vienen colocados de manera lógica entre una capa de alto nivel que consta de usuarios y aplicaciones, y una capa subyacente constituida por sistemas operativos y recursos básicos de comunicación, tal como vemos en la figura 1-1. De acuerdo con lo anterior, a dicho sistema distribuido se le conoce como **middleware**.

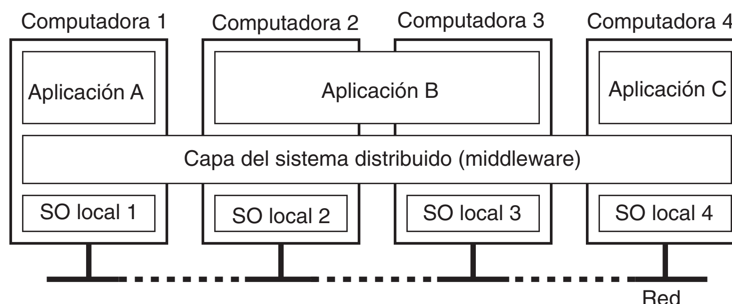


Figura 1-1. Un sistema distribuido organizado como middleware. La capa de middleware se extiende sobre diversas máquinas, y ofrece a cada aplicación la misma interfaz.

En la figura 1-1 podemos ver cuatro computadoras conectadas en red y tres aplicaciones, de las cuales la aplicación *B* está distribuida entre las computadoras 2 y 3. A cada aplicación se le ofrece la misma interfaz. El sistema distribuido proporciona los medios para que los componentes de una sola aplicación distribuida se puedan comunicar ente sí, pero también para permitir la comunicación entre las diferentes aplicaciones. Al mismo tiempo, oculta, lo mejor y más razonablemente posible, las diferencias que se presentan entre el hardware y los sistemas operativos para cada aplicación.

1.2 OBJETIVOS

Sólo porque es posible construir un sistema distribuido no significa que necesariamente sea una buena idea. Después de todo, con la tecnología actual también es posible colocar cuatro controladores de discos flexibles dentro de una computadora personal. Solamente que hacer eso no tendría sentido. En esta sección explicamos cuatro objetivos importantes que deben cumplirse para que la construcción de un sistema distribuido valga la pena. Un sistema distribuido debe hacer que los recursos sean fácilmente accesibles; debe ocultar de manera razonable el hecho de que los recursos están distribuidos por toda la red; debe ser abierto; y debe ser escalable.

1.2.1 Cómo hacer accesibles los recursos

El principal objetivo de un sistema distribuido es facilitar a los usuarios (y a las aplicaciones) el acceso a los recursos remotos, y compartirlos de manera controlada y eficiente. Los recursos pueden

significar casi cualquier cosa, pero ejemplos clásicos pueden ser impresoras, computadoras, dispositivos de almacenamiento, datos, archivos, páginas web, y redes, por nombrar algunos. Existen muchas razones para desear compartir recursos. Una razón evidente es la económica. Por ejemplo, es más barato permitir que una impresora sea compartida por distintos usuarios en una pequeña oficina que comprar y dar mantenimiento por separado a una impresora para cada usuario. De modo similar, tiene sentido económico compartir recursos costosos tales como supercomputadoras, sistemas de almacenamiento de alto rendimiento, o máquinas de composición tipográfica, entre otros periféricos de gran costo.

Conectar usuarios y recursos también facilita la colaboración y el intercambio de información, esto queda ilustrado por el éxito que ha tenido internet con su sencillo protocolo para permitir el intercambio de archivos, correo, documentos, audio, y video. Actualmente, la conectividad de internet está provocando la creación de numerosas organizaciones virtuales que trabajan juntas por medio de un **groupware**, esto es, un software para la edición colaborativa, la teleconferencia, y más. De modo similar, la conectividad en internet ha propiciado el comercio electrónico, lo cual nos permite comprar y vender todo tipo de bienes sin tener que ir físicamente a una tienda o incluso salir de casa.

Sin embargo, mientras la conectividad y el intercambio aumentan, la seguridad se vuelve cada vez más importante. En la práctica común, los sistemas proporcionan poca protección en contra del espionaje o de la intrusión en las comunicaciones. Con frecuencia, las claves y otra información sensible se envían como texto sin codificar (es decir, sin encriptar) a través de la red, o se almacenan en servidores donde sólo tenemos la esperanza de que estarán protegidos. En este sentido, existe mucho espacio para la mejora. Por ejemplo, normalmente es posible ordenar suministros proporcionando solamente el número de alguna tarjeta de crédito. Pocas veces se requiere probar que el cliente posee la tarjeta. En el futuro, registrar las órdenes de compra de esta manera será posible únicamente si usted prueba que, en efecto, tiene físicamente una tarjeta al insertarla dentro de un lector de tarjetas.

Otro problema de seguridad es el de rastrear la comunicación con el fin de construir un perfil de preferencias para un usuario específico (Wang y cols. 1998). Tal rastreo viola de manera explícita la privacidad, especialmente cuando se hace sin notificarle al usuario. Un problema relacionado es que el aumento de la conectividad puede provocar una comunicación no deseada, tal como el correo basura, con frecuencia llamado *spam*. En tales casos, lo que necesitamos es protegernos mediante el uso de filtros de información especiales que seleccionan los mensajes entrantes basándose en su contenido.

1.2.2 Transparencia en la distribución

Un objetivo importante de un sistema distribuido es ocultar el hecho de que sus procesos y recursos están físicamente distribuidos a través de múltiples computadoras. Decimos que un sistema distribuido es **transparente** si es capaz de presentarse ante los usuarios y las aplicaciones como si se tratara de una sola computadora. Primero echemos un vistazo a los tipos de transparencia que existen en los sistemas distribuidos. Después, enfrentaremos la pregunta general con respecto a si la transparencia es siempre requerida.

Tipos de transparencia

Es posible aplicar el concepto de transparencia a distintos aspectos de un sistema distribuido, la figura 1-2 muestra una relación de los tipos más importantes.

Transparencia	Descripción
Acceso	Oculta diferencias en la representación de los datos y la forma en que un recurso accede a los datos
Ubicación	Oculta la localización de un recurso
Migración	Oculta el que un recurso pudiera moverse a otra ubicación
Reubicación	Oculta el que un recurso pudiera moverse a otra ubicación mientra está en uso
Replicación	Oculta el número de copias de un recurso
Concurrencia	Oculta que un recurso puede ser compartido por varios usuarios que compiten por él
Falla	Oculta la falla y recuperación de un recurso

Figura 1-2. Distintas formas de transparencia aplicables en un sistema distribuido (ISO, 1995).

La **transparencia de acceso** se encarga de ocultar las diferencias en la representación de los datos y la manera en que el usuario accede a dichos recursos. En un nivel básico, queremos ocultar las diferencias en la arquitectura de las máquinas, pero es más importante llegar a un acuerdo con respecto a la manera en que representamos los datos en las diferentes máquinas y sistemas operativos. Por ejemplo, un sistema distribuido puede tener sistemas de cómputo que ejecutan distintos sistemas operativos, cada uno con su propia convención para descifrar la nomenclatura de los archivos. Las diferencias en la convención de nombres, así como la manera en que podemos manipular los archivos, deberán quedar ocultas para los usuarios y las aplicaciones.

Un grupo importante de tipos de transparencia tiene que ver con la ubicación de un recurso. La **transparencia de ubicación** se refiere al hecho de que los usuarios no pueden determinar en qué ubicación física se localiza el sistema. La nomenclatura juega un papel muy importante para lograr la transparencia. En especial, podemos lograr la transparencia de ubicación si asignamos solamente nombres lógicos a los recursos, esto es, nombres en los cuales la ubicación de un recurso no quede secretamente codificada. Un ejemplo de dicho nombre es la URL http://www.pearsoneducacion.net/index.html, la cual no proporciona indicio alguno con respecto a la ubicación del principal servidor web de Pearson Educación. Además, esta URL tampoco da ninguna clave con respecto a si *index.html* ha estado siempre en su ubicación actual o fue reubicada recientemente. Decimos que los sistemas distribuidos en los cuales es posible reubicar los recursos sin afectar la manera en que podemos acceder a dichos recursos proporcionan **transparencia de migración**. Es incluso más importante la situación en la cual podemos reubicar los recursos *mientras* accedemos a ellos sin que el usuario o la aplicación lo noten. En tales casos, decimos que el sistema permite una **transparencia de reubicación**. Un ejemplo de transparencia de reubicación es cuando los usuarios móviles pueden continuar usando sus computadoras portátiles inalámbricas mientras se mueven de un lugar a otro sin desconectarse (temporalmente).

Como veremos, la replicación juega un papel muy importante en los sistemas distribuidos. Por ejemplo, podemos replicar los recursos para incrementar la disponibilidad o para mejorar el rendi-

miento desde donde se acceda a ésta. La **transparencia de replicación** tiene que ver con el hecho de ocultar que existen distintas copias del recurso. Para ocultar la replicación a los usuarios, es necesario que todas las réplicas tengan el mismo nombre. En consecuencia, un sistema que da soporte a la transparencia de replicación generalmente debe sustentar también la transparencia de ubicación, porque de lo contrario sería imposible hacer referencia a réplicas localizadas en diferentes ubicaciones.

Acabamos de mencionar que un objetivo importante de los sistemas distribuidos es permitir el intercambio de recursos. En muchos casos, el intercambio de recursos se puede lograr de manera cooperativa, como en el caso de la comunicación. Sin embargo, también existen ejemplos de intercambio de recursos de manera competida. Por ejemplo, dos usuarios independientes pueden tener almacenados cada uno sus archivos en un mismo servidor de archivos, o en una base de datos compartida pudieran acceder a las mismas tablas. En tales casos, es importante que cada usuario no advierta que el otro usuario hace uso del mismo recurso. A este fenómeno se le llama **transparencia de concurrencia**. Un problema importante es que el acceso concurrente a un recurso compartido deja ese recurso en un estado consistente. La consistencia se puede alcanzar a través de mecanismos de bloqueo mediante los cuales se concede al usuario, por turno, el acceso exclusivo al recurso deseado. Un mecanismo más definido es hacer uso de transacciones, pero como veremos en capítulos posteriores, éstas son muy difíciles de implementar en sistemas distribuidos.

Una definición alternativa muy popular acerca de un sistema distribuido, la cual se debe a Leslie Lamport, es: "Usted se entera de que tiene un sistema distribuido cuando la falla de una computadora de la cual nunca había escuchado le impide realizar cualquier trabajo." Esta descripción pone al descubierto otro problema importante que se presenta durante el diseño de los sistemas distribuidos: lidiar con las fallas. Hacer que un sistema distribuido sea **transparente a fallas** significa lograr que el usuario no se percate de que un recurso (del cual quizá nunca oyó hablar) deja de funcionar correctamente, y que después el sistema se recupere de la falla. Enmascarar las fallas es uno de los problemas más difíciles de solucionar en los sistemas distribuidos, e incluso resulta imposible lograrlo cuando hacemos ciertas suposiciones aparentemente realistas, como veremos en el capítulo 8. La principal dificultad que se presenta en el enmascaramiento de las fallas radica en la falta de habilidad para distinguir entre un recurso muerto y un recurso penosamente lento. Por ejemplo, cuando hace contacto con un servidor web ocupado, el navegador genera un error (de *time out*) e indica que la página web no está disponible. En ese punto, el usuario no puede concluir si el servidor realmente se encuentra fuera de servicio.

Nivel de transparencia

Aunque la transparencia de distribución generalmente es considerada preferible para cualquier sistema distribuido, existen situaciones en las que tratar de ocultar por completo todos los aspectos de distribución a los usuarios no es una buena idea. Por ejemplo, cuando usted solicita que su periódico electrónico aparezca en su cuenta de correo antes de las 7 A.M. tiempo local, como es lo normal, mientras que usted se encuentra al otro lado del mundo y vive en una zona horaria completamente distinta. Su periódico matutino no será el periódico matutino al que usted está acostumbrado.

De manera similar, no podemos esperar que un sistema distribuido de área amplia que conecta un proceso ubicado en San Francisco a un proceso en Amsterdam oculte el hecho de que la madre naturaleza no permita enviar un mensaje de un proceso a otro a menos de unos 35 milisegundos. En la práctica, toma varios cientos de milisegundos al usar una red de computadoras. La transmisión de señal no solamente está limitada por la velocidad de la luz, sino también por las capacidades de los interruptores intermedios.

También existe cierto intercambio entre un alto grado de transparencia y el rendimiento del sistema. Por ejemplo, muchas aplicaciones de internet tratan repetidamente de contactar a un servidor antes de darse por vencidas. En consecuencia, intentar enmascarar la falla de un servidor transitorio antes de buscar el contacto con otro servidor, pudiera volver más lento a todo el sistema. En tal caso, podría ser preferible desistir antes, o al menos permitir que el usuario intente hacer contacto.

Otro ejemplo es cuando necesitamos garantizar que las diversas réplicas, localizadas en diferentes continentes, sean consistentes (coherentes) todo el tiempo. En otras palabras, si modificamos una copia, ese cambio se debe propagar a todas las copias antes de permitir cualquier otra operación. Queda claro que una sencilla operación de actualización podría no tomar ni siquiera un segundo para llevarse a cabo, algo que no podemos ocultar de los usuarios.

Por último, existen situaciones donde no resulta muy evidente que ocultar la distribución sea buena idea. Conforme los sistemas distribuidos se expanden hacia los dispositivos que los usuarios traen consigo (móviles y ubicuos), y donde la simple noción de la ubicación y de la idea de contexto se vuelve cada vez más importante, podría ser mejor *exponer* la distribución en lugar de tratar de ocultarla. Esta exposición de la distribución se hará más evidente cuando estudiemos los sistemas distribuidos embebidos (integrados) y ubicuos, más adelante en este capítulo. Como un ejemplo sencillo, considere a un empleado de oficina que quiere imprimir un archivo desde su computadora personal. Es mejor enviar el trabajo de impresión a una impresora ocupada, pero cercana, en lugar de enviarlo a alguna impresora libre pero localizada en las oficinas corporativas en otro país.

Existen otros argumentos en contra de la transparencia de distribución. Reorganizar por completo la transparencia de distribución es simplemente imposible, incluso debemos preguntarnos si es inteligente *pretender* que podemos lograrla. Podría ser mucho mejor hacer una distribución explícita de modo que nunca engañemos al desarrollador de la aplicación ni al usuario para que crean que existe algo como la transparencia. El resultado será que los usuarios comprenderán mucho mejor el (a veces inesperado) comportamiento de un sistema distribuido, y estarán entonces mucho mejor preparados para interactuar con ese comportamiento.

La conclusión es que buscar la transparencia de distribución puede ser un buen objetivo cuando diseñamos e implementamos sistemas distribuidos, pero debemos considerarla junto con otros problemas tales como el rendimiento y la comprensibilidad. El costo de no tener capacidad para lograr la transparencia completa puede ser sorprendentemente alto.

1.2.3 Grado de apertura

Otro objetivo importante de los sistemas distribuidos es el grado de apertura. Un **sistema distribuido abierto** es un sistema que ofrece servicios de acuerdo con las reglas estándar que describen la sintaxis y la semántica de dichos servicios. Por ejemplo, en las redes de computadoras, las reglas

estándar gobiernan formato, contenido, y significado de los mensajes enviados y recibidos. Tales reglas se formalizan mediante protocolos. En los sistemas distribuidos, por lo general, los servicios se especifican a través de **interfaces**, las cuales a menudo se definen como **lenguaje de definición de interfaz** (**IDL**, por sus siglas en inglés). Por lo general, las definiciones de interfaz escritas en IDL solamente capturan la sintaxis de los servicios. En otras palabras, especifican de manera precisa los nombres de las funciones disponibles junto con los tipos de parámetros, valores de retorno, posibles excepciones que se pueden alcanzar, entre otros elementos. La parte difícil es especificar de modo preciso lo que hacen esos servicios, esto es, la semántica de las interfaces. En la práctica, tales especificaciones siempre están dadas de manera informal por medio de un lenguaje natural.

Cuando la especificamos correctamente, una definición de interfaz permite iniciar un proceso arbitrario que requiere cierta interfaz para comunicarse con otro proceso que proporcione tal interfaz. También permite que dos componentes independientes construyan implementaciones completamente distintas a partir de dichas interfaces, lo cual da lugar a dos sistemas distribuidos separados que operan exactamente de la misma forma. Las especificaciones apropiadas son completas y neutrales. Completo significa que todo lo necesario para efectuar una implementación ha quedado especificado. Sin embargo, muchas definiciones de interfaz no están del todo completas, de modo que para un desarrollador es necesario añadir detalles específicos de la implementación. Igual de importante es el hecho de que las especificaciones no describen la manera en que se deben ver; deberán ser neutrales. Que las definiciones de las interfaces sean completas y neutrales es un factor muy importante para lograr la interoperabilidad y portabilidad (Blair y Stefani, 1998). La **interoperabilidad** define la extensión mediante la cual dos implementaciones de sistemas o componentes de fabricantes distintos pueden coexistir y trabajar juntos si únicamente se apoyan en sus servicios mutuos tal como se especifica mediante un estándar común. La **portabilidad** define la extensión mediante la cual una aplicación desarrollada para un sistema distribuido *A* se pueda ejecutar, sin modificación, en un sistema distribuido *B* que comparte la misma interfaz que *A*.

Otro objetivo importante para un sistema distribuido abierto es que debiera ser más fácil configurar el sistema para componentes diferentes (quizá de distintos desarrolladores). Además, debiera ser más fácil agregar nuevos componentes o reemplazar los existentes sin afectar aquellos que permanecen en su lugar. En otras palabras, un sistema distribuido abierto debe ser también **extensible**. Por ejemplo, en un sistema extensible, debiera ser relativamente fácil agregar partes que se ejecutan en sistemas operativos diferentes, o incluso reemplazar todo un sistema de archivos. Como sabemos a partir de la práctica diaria, dicha flexibilidad es más fácil de decir que de alcanzar.

Separar la política de la mecánica

Para lograr la flexibilidad en los sistemas distribuidos abiertos, es crucial que estén organizados como una colección de componentes relativamente pequeños y fáciles de reemplazar o adaptar. Esto implica que debiéramos proporcionar definiciones no solamente para las interfaces de más alto nivel, esto es, aquellas que las aplicaciones y los usuarios ven, sino también las definiciones implementadas para las interfaces internas del sistema y describir la manera en que éstas interactúan. Este enfoque es relativamente nuevo. Muchos sistemas antiguos, e incluso contemporáneos, están cons-

truidos mediante la utilización de un método monolítico en el cual los componentes se separan sólo de manera lógica, pero son implementados como un único y enorme programa. Este método vuelve más difícil el reemplazo o la adaptación de componentes sin que resulte afectado todo el sistema. De esta manera, los sistemas monolíticos tienden a ser cerrados en lugar de abiertos.

Con frecuencia, la necesidad de modificar un sistema distribuido es ocasionada por un componente que no proporciona la política óptima para un usuario o una aplicación en específico. Como ejemplo, considere el manejo del caché dentro de la World Wide Web. Por lo general, los navegadores permiten al usuario adaptar su política de manejo de caché al especificar el tamaño del caché, y se debe o no verificar siempre la consistencia de un documento que utiliza caché, o quizá solamente una vez por sesión. Sin embargo, el usuario no puede modificar otros parámetros del uso de caché, tales como el tiempo que un documento debe permanecer en el caché, o cuál documento debe retirarse cuando el caché se llena. Además, es imposible tomar decisiones con respecto al uso del caché basadas en el *contenido* de un documento. Por ejemplo, un usuario pudiera insertar en el caché itinerarios de los trenes, si sabe que casi nunca sufren modificaciones, pero nunca guardar información sobre las condiciones del tráfico en las carreteras.

Lo que necesitamos es implementar una separación entre la política y la mecánica. Por ejemplo, en el caso del uso de caché en la web, el navegador debiera proporcionar las facilidades apropiadas para almacenar solamente documentos, y al mismo tiempo permitir a los usuarios decidir cuáles documentos se almacenan y por cuánto tiempo. En la práctica, podemos implementar esto mediante la oferta de todo un conjunto de parámetros que el usuario puede personalizar (de manera dinámica). Incluso es mejor si el usuario puede implementar su propia política en la forma de un componente que sea posible conectar dentro del navegador. Por supuesto, dicho componente debe tener una interfaz que el usuario pueda comprender de manera que pueda llamar procedimientos de dicha interfaz.

1.2.4 Escalabilidad

La conectividad a nivel mundial a través de internet se está haciendo tan común como enviar una postal a cualquier persona que se encuentre en cualquier parte del mundo. Con esto en mente, para los desarrolladores de sistemas distribuidos, la escalabilidad es uno de los objetivos más importantes.

La escalabilidad de un sistema se puede medir de acuerdo con al menos tres dimensiones (Neuman, 1994). Primero, un sistema puede ser escalable con respecto a su tamaño, lo cual significa que podemos agregarle fácilmente usuarios y recursos. Segundo, un sistema escalable geográficamente es aquel en el cual usuarios y recursos pueden radicar muy lejos unos de los otros. Tercero, un sistema puede ser escalable administrativamente; esto es, puede ser fácil de manejar incluso si involucra muchas organizaciones administrativas diferentes. Desafortunadamente, con frecuencia un sistema escalable en una o más de estas dimensiones exhibe alguna pérdida de rendimiento al escalarlo.

Problemas de escalabilidad

Cuando un sistema requiere ser escalado, hay que resolver muchos tipos de problemas diferentes. Consideremos primero la escalabilidad relativa al tamaño. Si más usuarios o recursos requieren

soporte, con frecuencia se enfrentan limitaciones en los servicios, datos, y algoritmos centralizados (vea la figura 1-3). Por ejemplo, muchos servicios están centralizados en el sentido de que se implementan en términos de un solo servidor que se ejecuta en una máquina específica localizada en el sistema distribuido. El problema con este esquema resulta evidente: el servidor se puede convertir en un cuello de botella mientras el número de usuarios y aplicaciones crece. Incluso si contamos con una capacidad de almacenamiento y procesamiento ilimitada, en algún momento la comunicación con el servidor prohibirá cualquier crecimiento posterior.

Por desgracia, en algunas ocasiones no podemos evitar el uso de un solo servidor. Imagine que tenemos un servicio para manipular información confidencial tal como expedientes médicos, cuentas de banco, y así por el estilo. En tales casos, pudiera ser mejor implementar dicho servicio por medio de un solo servidor en una ubicación por separado de alta seguridad, y protegida por otras partes del sistema distribuido a través de componentes de red especiales. Copiar el servidor hacia diversas ubicaciones para aumentar el rendimiento podría estar fuera de contexto ya que volvería menos seguro el servicio.

Concepto	Ejemplo
Servicios centralizados	Un solo servidor para todos los usuarios
Datos centralizados	Un solo directorio telefónico en línea
Algoritmos centralizados	Hacer ruteo basado en información completa

Figura 1-3. Ejemplos de limitaciones en la escalabilidad.

Algo tan malo como los servicios centralizados son los datos centralizados. ¿Cómo podemos seguir la pista de los números telefónicos y las direcciones de 50 millones de personas? Suponga que cada registro cabe dentro de 50 caracteres. Una sola partición de disco de 2.5 gigabytes podría proporcionar espacio suficiente. Pero, de nuevo, tener una sola base de datos sin duda saturará todas las líneas de comunicación que vayan hacia y desde la base de datos. De manera similar, imagine la manera en que internet funcionaría si su servicio de nombres de dominio (DNS, por sus siglas en inglés) estuviera implementado dentro de una sola tabla. El DNS mantiene información en millones de computadoras ubicadas alrededor del mundo y constituye un servicio esencial para localizar los servidores web. Si cada petición para resolver una URL debiera reenviarse a un servidor DNS único, resulta evidente que nadie estaría utilizando la web (lo cual, por cierto, resolvería el problema).

Por último, los algoritmos centralizados también son una mala idea. En sistemas distribuidos grandes, hay que rutear un enorme número de mensajes a lo largo de muchas líneas. Desde un punto de vista teórico, la manera óptima de hacer esto es recopilar toda la información acerca de la carga en todas las máquinas y líneas, y posteriormente ejecutar un algoritmo para calcular todas las rutas óptimas. Podemos diseminar esta información a lo largo del sistema para mejorar el ruteo.

El problema es que recopilar y transportar toda la información de entrada y salida sería de nuevo una mala idea debido a que estos mensajes sobrecargan parte de la red. De hecho, debemos evitar cualquier algoritmo que opere recopilando información desde todos los sitios, enviándola a una

sola máquina para procesarla, y distribuyendo luego los resultados. Deberíamos utilizar solamente algoritmos descentralizados. Por lo general, estos algoritmos cuentan con las siguientes características, las cuales los distinguen de los algoritmos centralizados:

1. Ninguna máquina tiene información completa con respecto al estado del sistema.
2. Las máquinas toman decisiones con base en la información local.
3. La falla de una sola máquina no arruina todo el algoritmo.
4. No existen suposiciones implícitas con respecto a la existencia de un reloj global.

Las tres primeras características se derivan de lo que ya mencionamos hasta aquí. La última es quizá menos evidente, pero no menos importante. Cualquier algoritmo que comience con: "Exactamente a las 12:00:00 todas las máquinas deben observar el tamaño de su cola de salida" fallará debido a que es imposible obtener la sincronización de todos los relojes. Los algoritmos deben tomar en cuenta la falta de sincronización entre los relojes involucrados. Mientras más grande sea el sistema, más grande será la incertidumbre. En una sola LAN, mediante un esfuerzo considerable podría ser posible obtener la sincronización de todos los relojes hasta unos cuantos microsegundos, pero hacer esto a nivel nacional o internacional es imposible.

La escalabilidad geográfica tiene sus propios problemas. Una de las razones por las cuales es difícil escalar sistemas distribuidos existentes diseñados para redes de área local es que se basan en la **comunicación síncrona**. En esta forma de comunicación, una parte que solicita un servicio, por lo general llamada **cliente**, bloquea el servicio hasta que obtiene una respuesta. Por lo común, este método funciona bien en redes de área local donde la comunicación entre dos máquinas es, en el peor de los casos, de unos cuantos cientos de microsegundos. Sin embargo, dentro de una red de área amplia, necesitamos tomar en cuenta que la comunicación interproceso puede ser de cientos de milisegundos, tres veces más pequeña en magnitud. Construir aplicaciones interactivas mediante el uso de comunicación síncrona en sistemas de área amplia requiere mucho cuidado (y no poca paciencia).

Otro problema que obstaculiza la escalabilidad geográfica es que en redes de área amplia la comunicación es inherentemente no fiable, y virtualmente siempre es de punto a punto. Por el contrario, generalmente las redes de área local proporcionan facilidades de comunicación basadas en difusión altamente confiable, lo cual vuelve más fácil desarrollar sistemas distribuidos. Por ejemplo, considere el problema de localizar un servicio. En un sistema de área local, un proceso puede sencillamente difundir un mensaje a cada máquina, preguntando si se ejecuta el servicio requerido. Solamente las máquinas que hacen que dicho servicio responda proporcionan cada una su dirección de red en el mensaje de respuesta. Dicho esquema de ubicación es impensable en un sistema de área amplia: solamente imagine lo que pasaría si queremos localizar de esta manera un servicio en internet. En lugar de lo anterior, necesitamos diseñar servicios especiales de ubicación, los cuales podrían necesitar ser escalados a nivel mundial y tener capacidad de servir a millones de usuarios. Regresaremos a dichos servicios en el capítulo 5.

La escalabilidad geográfica está relacionada íntimamente con los problemas referentes a las soluciones centralizadas que obstaculizan la escalabilidad de tamaño. Si tenemos un sistema con muchos componentes centralizados, es evidente que la escalabilidad geográfica quedará limitada por los problemas de rendimiento y confiabilidad que resultan a partir de la comunicación de área

amplia. Además, los componentes centralizados provocan el desperdicio de servicios de red. Imagine que utilizamos un solo servidor de correo para todo un país. Esto significaría que para enviar un mensaje a su vecino el correo tendría que ir primero a un servidor de correo central, el cual pudiera estar a miles de kilómetros de distancia. Claramente, ésta no es una forma correcta de implementación.

Por último, tenemos una dificultad, y en muchos casos la pregunta abierta es cómo escalar un sistema a través de múltiples dominios administrativamente independientes. Un problema importante a solucionar es el relacionado con políticas conflictivas con respecto al uso de los recursos (y pagos), a la administración, y a la seguridad.

Por ejemplo, con frecuencia muchos componentes de un sistema distribuido que reside dentro de un solo dominio pueden ser asociados mediante los usuarios que operan dentro del mismo dominio. En tales casos, los sistemas de administración pueden haber probado y certificado las aplicaciones, y pudieran tomarse medidas especiales para asegurar que dichos componentes no puedan ser forzados. En esencia, los usuarios confían en sus administradores de sistemas. Sin embargo, esta confianza no se expande en forma natural a lo largo de los límites del dominio.

Si expandimos un sistema distribuido dentro de otro dominio, debemos tomar dos tipos de medidas de seguridad. Primero, el sistema distribuido debe autoprotegerse en contra de ataques maliciosos provenientes del nuevo dominio. Por ejemplo, los usuarios del nuevo dominio pudieran tener acceso de sólo lectura hacia el sistema de archivos en su dominio original. De manera similar, recursos tales como costosas máquinas de composición de imagen o computadoras de alto rendimiento pudieran no estar disponibles para los usuarios externos. Segundo, el nuevo dominio tiene que protegerse contra ataques maliciosos provenientes del sistema distribuido. Un ejemplo típico es la descarga de programas tales como *applets* en los navegadores web. De manera básica, el nuevo dominio no sabe qué comportamiento esperar de dicho código externo, y por tanto pudiera decidir limitar de manera severa los derechos de acceso para dicho código. El problema, como veremos en el capítulo 9, es cómo reforzar dichas limitaciones.

Técnicas de escalamiento

Ya que explicamos algunos problemas de escalabilidad, surge la pregunta de cómo podremos resolverlos de manera general. En la mayoría de los casos, los problemas de escalabilidad en sistemas distribuidos aparecen como problemas de rendimiento ocasionados por la limitación de capacidad de servidores y redes. Por ahora, existen básicamente sólo tres técnicas para efectuar el escalamiento: ocultar las latencias de comunicación, distribución y replicación [vea también Neuman (1994)].

Ocultar las latencias de comunicación es importante para lograr la escalabilidad geográfica. La idea básica es simple: intentar evitar lo más posible la espera por respuestas de peticiones remotas (y potencialmente distantes) de servicios. Por ejemplo, cuando requerimos un servicio de una máquina remota, una alternativa durante la espera de respuesta del servidor es hacer otras cosas útiles del lado de la máquina que realiza la petición. En esencia, esto significa la construcción de la aplicación que hace las solicitudes de tal manera que solamente utilice **comunicación asíncrona**. Cuando obtenemos respuesta, interrumpimos la aplicación e invocamos un manejador (handler) especial para completar la petición hecha con anterioridad. A menudo es posible utilizar la comu-

nicación asíncrona para completar la petición hecha con anterioridad. Con frecuencia, la comunicación asíncrona se utiliza en sistemas de procesamiento por lotes y en aplicaciones paralelas, en las cuales se pueden calendarizar más o menos tareas independientes para ser ejecutadas mientras otra tarea espera para completar su comunicación. De manera alternativa, podemos iniciar un nuevo hilo de control para realizar la petición. Aunque se realiza un bloqueo mientras se espera la respuesta, podemos continuar con la ejecución de otros hilos presentes en el proceso.

Sin embargo, existen muchas aplicaciones que no pueden utilizar de manera efectiva la comunicación asíncrona. Por ejemplo, en aplicaciones interactivas, cuando un usuario envía una petición por lo general no tiene nada mejor que hacer que esperar por la respuesta. En tales casos, una mejor solución es reducir el volumen general de la comunicación; por ejemplo, si movemos parte del cálculo que normalmente hace el servidor hacia el proceso del cliente que hace la petición. Un caso típico en donde este método funciona es el acceso a bases de datos mediante el uso de formas. El llenado de las formas se puede hacer enviando mensajes por separado para cada campo, y esperando el reconocimiento de parte del servidor, como podemos ver en la figura 1-4(a). Por ejemplo, el servidor puede verificar errores de sintaxis antes de aceptar la entrada. Una mejor solución es enviar el código para llenar la forma, y posiblemente verificar las entradas, hacia el cliente, y que éste devuelva una forma completa, como vemos en la figura 1-4(b). Actualmente, este método de envío de código está ampliamente soportado por la web en la forma de applets de Java Javascript.

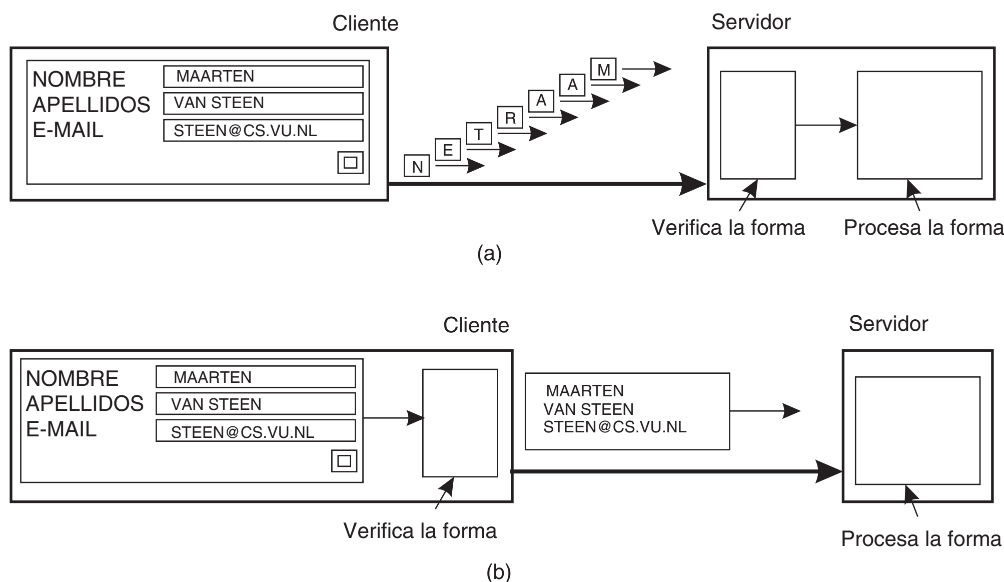


Figura 1-4. Diferencia entre dejar que (a) un servidor o (b) un cliente verifique formas mientras las llena.

Otra importante técnica de escalamiento es la **distribución**. La distribución significa tomar un componente, dividirlo en partes más pequeñas, y en consecuencia dispersar dichas partes a lo largo

del sistema. Un ejemplo excelente de distribución es el servicio de nombres de dominio usado en internet. El espacio de nombres de DNS está organizado de manera jerárquica dentro de tres **dominios**, los cuales están divididos en **zonas**, como podemos ver en la figura 1-5. Los nombres colocados en cada zona se manejan mediante un solo servidor de nombres. Sin entrar en más detalles, puede pensarse en cada nombre de ruta como el nombre de un servidor (*host)* en internet, y por tanto que está asociado con una dirección de red de dicho servidor. Básicamente, resolver un nombre significa devolver la dirección de red del servidor asociado. Considere, por ejemplo, el nombre *nl.vu.cs.flits*. Para resolver este nombre, primero se pasa al servidor de la zona *Z1* (vea la figura 1-5), la cual devuelve la dirección del servidor para la zona *Z2*, el cual puede manipular el resto del nombre *vu.cs.flits*. El servidor para *Z2* devolverá la dirección del servidor para la zona *Z3*, que es capaz de manejar la última parte del nombre y devolverá la dirección del servidor asociado.

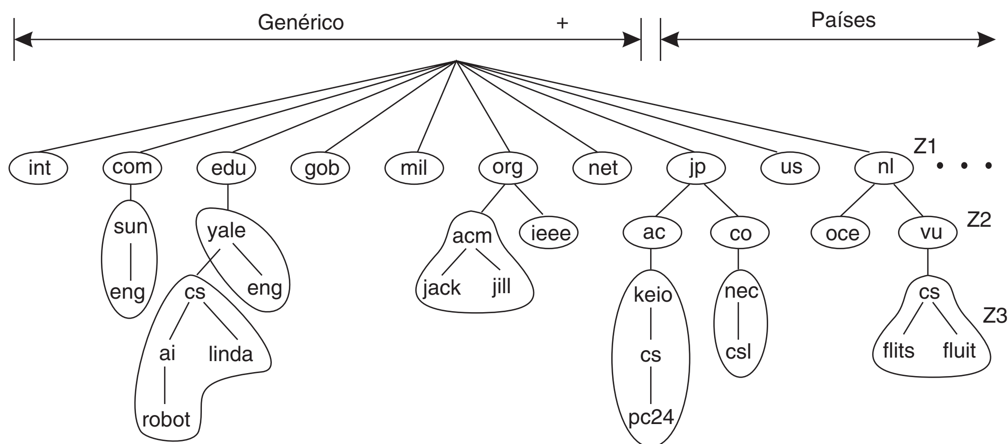


Figura 1-5. Ejemplo de cómo dividir el espacio de nombre DNS en zonas.

Este ejemplo muestra la forma en que el *servicio de nombres*, como el proporcionado por DNS, es distribuido a través de distintas máquinas, y de este modo se evita que un solo servidor tenga que lidiar con todas las peticiones para efectuar la resolución de nombres.

Como ejemplo adicional, considere la World Wide Web. Para la mayoría de los usuarios, la web pareciera ser un enorme sistema de información basado en documentos, donde cada documento contiene un nombre único en forma de una URL. Conceptualmente, podría incluso parecer como si fuera un solo servidor. Sin embargo, la web está distribuida físicamente a través de un gran número de servidores, y cada servidor maneja cierta cantidad de documentos web. El nombre del servidor que maneja el documento está codificado dentro de la URL de dicho documento. Debido a esta distribución de los documentos, la web es capaz de escalarlos a su tamaño actual.

Si consideramos que los problemas de escalabilidad aparecen con frecuencia en forma de degradación del rendimiento, por lo general es una buena idea **replicar** los componentes a lo largo del

sistema distribuido. La replicación no solamente incrementa la disponibilidad, sino que además ayuda a balancear la carga entre los componentes para obtener un mejor rendimiento. Además, en los sistemas geográficos dispersados ampliamente, tener una copia cercana puede ocultar muchos de los problemas de latencia de comunicación mencionados anteriormente.

Hacer uso del **caché** es una forma especial de replicación, aunque con frecuencia la distinción entre los dos procedimientos es difícil de realizar o incluso de fabricar artificialmente. Como en el caso de la replicación, el caché provoca la elaboración de una copia, por lo general en la proximidad del cliente que accede a dicho recurso. Sin embargo, en contraste con la replicación, el caché es una decisión tomada por el cliente de un recurso, y no por el propietario del recurso. Además, el caché sucede sobre demanda en donde la replicación a menudo está planeada por adelantado.

Existe una seria desventaja en el uso del caché y la replicación que pudiera afectar seriamente la escalabilidad. Debido a que por ahora tenemos copias múltiples de un recurso, modificar una copia la hace diferente del resto. En consecuencia, el caché y la replicación provocan problemas de **consistencia** (incoherencia en la información).

Hasta qué punto pueden tolerarse las inconsistencias, depende en gran medida del uso de un recurso. Por ejemplo, muchos usuarios de la web consideran aceptable que su navegador devuelva un documento almacenado en caché cuya verificación de validez no ha sido confirmada durante el último par de minutos. Sin embargo, existen muchos casos en los que se requiere cumplir con una fuerte garantía de consistencia, tal como en las transacciones electrónicas de inventario y subastas. El fuerte problema con la consistencia es que una actualización se debe propagar de inmediato a todas las demás copias. Más aún, cuando suceden dos actualizaciones de manera concurrente, a menudo se requiere que cada copia se actualice en el mismo orden. Situaciones como éstas requieren, por lo general, un mecanismo de sincronización global. Desafortunadamente, dichos mecanismos son en extremo difíciles o incluso imposibles de implementar de manera escalable, pues se insiste en que los fotones y las señales eléctricas obedecen a un límite de velocidad de 187 millas/ms (301 km/milisegundo) (la velocidad de la luz). Como consecuencia, el escalamiento mediante replicación podría introducir otras soluciones inherentes y no escalables. Regresaremos a la replicación y la consistencia en el capítulo 7.

Cuando consideramos estas técnicas de escalamiento, podríamos argumentar que, desde el punto de vista técnico, el tamaño de la escalabilidad es lo menos problemático. En muchos casos, simplemente al incrementar la capacidad de una máquina salvará el día (al menos temporalmente y quizás a un costo significativo). La escalabilidad geográfica es un problema mucho más serio con la intromisión de la madre naturaleza. Sin embargo, con frecuencia la práctica muestra que si combinamos la distribución, la replicación, y las técnicas de uso del caché con formas diferentes de consistencia tendremos suficiente en la mayoría de los casos. Finalmente, la escalabilidad en la administración parece ser la que presenta mayor dificultad, en parte debido a que necesitamos resolver problemas no técnicos (es decir, las políticas de una empresa y la colaboración humana). Sin embargo, se han logrado progresos en esta área; esto es, simplemente se *ignoran* los dominios administrativos. La introducción y la manera en que se dispersa el uso de la tecnología de igual a igual demuestra lo que puede lograrse cuando los usuarios finales toman el control (Aberer y Hauswirth, 2005; Lua *et al.*, 2005; y Oram, 2001). No obstante, dejemos en claro que la tecnología de punto a punto —en el mejor de los casos— solamente puede tomarse como una solución parcial para resolver la escalabilidad administrativa. En algún momento deberemos lidiar con ella.

1.2.5 Trampas

En este punto, debemos tener claro ya que el desarrollo de sistemas distribuidos puede ser una tarea formidable. Como veremos muchas veces a lo largo del libro, existen tantos problemas a considerar al mismo tiempo que solamente la complejidad puede ser el resultado. Sin embargo, mediante el seguimiento de cierto número de principios de diseño, los sistemas distribuidos se pueden desarrollar y adherir fuertemente a los objetivos establecidos en este capítulo. Muchos principios siguen las reglas básicas de la ingeniería de software decente y no los repetiremos aquí.

Sin embargo, los sistemas distribuidos difieren del software tradicional debido a que sus componentes están dispersos por toda una red. No tomar en cuenta esta dispersión durante la etapa de diseño es lo que vuelve demasiado complejos a muchos sistemas innecesariamente y provoca errores que, a la larga, se tienen que reparar. Peter Deutsch, en aquel entonces de Sun Microsystems, formuló estos errores como las siguientes falsas suposiciones que todos hacemos al desarrollar por primera vez un sistema distribuido:

1. La red es confiable.
2. La red es segura.
3. La red es homogénea.
4. La topología no cambia.
5. La latencia es igual a cero.
6. El ancho de banda es infinito.
7. El costo de transporte es igual a cero.
8. Existe un administrador.

Observe cómo estas suposiciones están relacionadas con las propiedades que son únicas para los sistemas distribuidos: confiabilidad, seguridad, heterogeneidad, y topología de la red; latencia y ancho de banda; costos de transporte; y finalmente dominios administrativos. Cuando desarrollamos aplicaciones no distribuidas, muchos de estos problemas no son detectados.

La mayor parte de los principios que explicamos en el libro se relacionan de inmediato con estas suposiciones. En todos los casos, explicaremos soluciones a problemas ocasionados por el hecho de que una o más de las suposiciones es falsa. Por ejemplo, las redes confiables no existen, lo cual provoca la imposibilidad de lograr la transparencia a fallas. Dedicamos un capítulo completo para tratar con el hecho de que la comunicación vía red es insegura por naturaleza. Ya argumentamos que los sistemas distribuidos requieren tomar en cuenta la heterogeneidad. En la misma línea, cuando explicamos la replicación para resolver los problemas de escalabilidad, esencialmente estamos atacando los problemas de latencia y ancho de banda. Tocaremos también problemas relacionados con la administración en distintos puntos a lo largo del libro, tratando con las falsas suposiciones del costo cero del transporte y un dominio administrativo individual.

1.3 TIPOS DE SISTEMAS DISTRIBUIDOS

Antes de explicar los principios de los sistemas distribuidos, primero demos un vistazo a los diversos tipos de sistemas distribuidos. A continuación señalamos la diferencia entre los distintos tipos de sistemas distribuidos de cómputo, sistemas distribuidos de información, y sistemas distribuidos embebidos.

1.3.1 Sistemas distribuidos de cómputo

Una clase importante de sistemas distribuidos es la utilizada para realizar tareas de cómputo de alto rendimiento. Hablando claro, podemos hacer una distinción entre dos subgrupos. En el **cómputo en cluster**, el hardware subyacente consta de una colección de estaciones de trabajo similares, o computadoras personales, conectadas cercanamente por medio de una red de área local de alta velocidad. Además, cada nodo ejecuta el mismo sistema operativo.

La situación se torna bastante diferente en el caso del **cómputo en malla (grid)**. Este subgrupo consta de sistemas distribuidos construidos generalmente como un conjunto de sistemas de cómputo, en donde cada sistema podría caer dentro de un dominio administrativo diferente, y podría ser muy diferente cuando nos referimos al hardware, software, y la tecnología de red instalada.

Sistemas de cómputo en cluster

Los sistemas de cómputo en cluster adquirieron popularidad cuando mejoró la relación precio-rendimiento de las computadoras personales y las estaciones de trabajo. Hasta cierto punto, tanto financiera como técnicamente se volvió atractivo construir una supercomputadora usando tecnología directamente del estante mediante la simple conexión de una colección de computadoras sencillas ubicadas dentro de una red de alta velocidad. En virtualmente todos los casos, la computación en cluster se utiliza para la programación en paralelo donde un solo programa (de cálculo intensivo) corre paralelamente en múltiples máquinas.

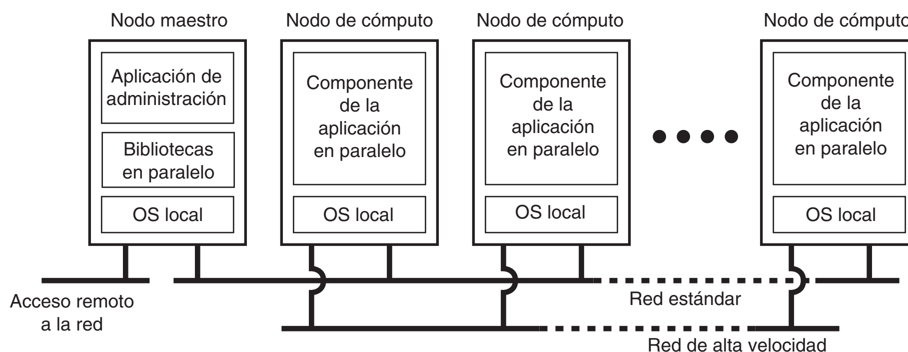


Figura 1-6. Ejemplo de un sistema de cómputo en cluster.

Un ejemplo muy conocido de una computadora en cluster es la formada por clústeres basados en Linux-Beowulf, en la figura 1-6 aparece su configuración general. Cada cluster consta de una colección de nodos de cómputo controlados y se accede a ellos mediante un solo nodo maestro. Por lo general, el nodo maestro manipula la ubicación de los nodos para un programa paralelo en particular, mantiene una cola de procesamiento por lotes de trabajos enviados, y proporciona una interfase para los usuarios del sistema. Como tal, el nodo maestro en realidad ejecuta el middleware necesario para la ejecución de programas y la administración del cluster, mientras que con frecuencia los nodos de cómputo no requieren nada más que un sistema operativo.

Una parte importante de este middleware está formada por las bibliotecas necesarias para ejecutar programas paralelos. Como veremos en el capítulo 4, muchas de estas bibliotecas proporcionan de manera efectiva sólo facilidades de comunicación avanzada basada en mensajes, pero no son capaces de manipular procesos con fallas, seguridad, etcétera.

Como alternativa para esta organización jerárquica, en el sistema MOSIX existe un método simétrico (Amar y cols., 2004). MOSIX intenta proporcionar una **imagen de sistema único** de un cluster, lo cual significa que la computadora en cluster ofrece al proceso la mayor transparencia de distribución al aparentar ser una sola computadora. Como ya mencionamos, es imposible proporcionar tal imagen bajo todas las circunstancias. En el caso de MOSIX, proporciona el grado de transparencia más alto al permitir a los procesos migrar de manera dinámica y preferencial entre los nodos que conforman el cluster. La migración de procesos permite a un usuario comenzar la aplicación en cualquier nodo (conocido como nodo de inicio), después de lo cual es posible realizar una migración hacia otros nodos, por ejemplo, para hacer más eficiente el uso de los recursos. Regresaremos a la migración de procesos en el capítulo 3.

Sistemas de cómputo en grid

Una característica del cómputo basado en cluster es su homogeneidad. En la mayoría de los casos, las computadoras en cluster son en esencia las mismas, todas tienen el mismo sistema operativo, y están conectadas a través de la misma red. Por el contrario, los sistemas de cómputo basados en grid tienen un alto grado de heterogeneidad: no se hacen suposiciones de ninguna índole con respecto al hardware, sistemas operativos, redes, dominios administrativos, políticas de seguridad, etcétera.

Una cuestión clave en un sistema de cómputo en grid es reunir los recursos de diferentes organizaciones para permitir la colaboración de un grupo de personas o instituciones. Tal colaboración se realiza en la forma de una **organización virtual**. La gente que pertenece a la misma organización virtual tiene derechos de acceso a los recursos que proporciona la organización. Por lo general, los recursos constan de servidores (incluso supercomputadoras, posiblemente implementadas como clústeres de computadoras), facilidades de almacenamiento, y bases de datos. Además, también es posible agregar dispositivos especiales de red como telescopios, sensores, etcétera.

Dada su naturaleza, la mayor parte del software para elaborar un sistema de cómputo en grid evoluciona alrededor del otorgamiento de privilegios a los recursos desde diferentes dominios administrativos, y solamente para aquellos usuarios y aplicaciones que pertenecen a una organización

virtual específica. Por esta razón, con frecuencia la atención está puesta en asuntos de arquitectura. En la figura 1-7 mostramos la arquitectura propuesta por Foster y colaboradores (2001).

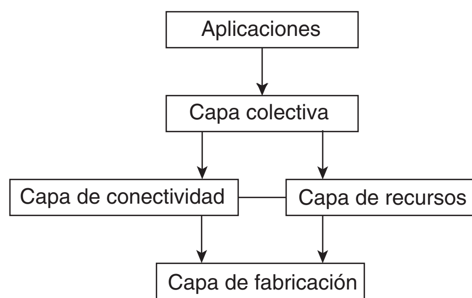


Figura 1-7. Arquitectura en capas para sistemas de cómputo en grid.

La arquitectura está compuesta por cuatro capas. La capa más baja, de *fabricación*, proporciona interfaces para recursos locales ubicados en un sitio específico. Observe que estas interfaces están adaptadas para permitir el intercambio de recursos dentro de una organización virtual. Por lo general, proporcionan funciones para consultar el estado y las capacidades de un recurso, junto con funciones para la administración real de un recurso (por ejemplo, bloqueo de recursos).

La *capa de conectividad* consiste en protocolos de comunicación para dar soporte a las transacciones del grid que abarcan el uso de múltiples recursos. Por ejemplo, los protocolos son necesarios para transferir datos entre los recursos, o simplemente para acceder a un recurso desde una ubicación remota. Además, la capa de conectividad contiene protocolos de seguridad para autentificar a usuarios y recursos. Observe que en muchos casos los usuarios humanos no se autentifican; en vez de eso, se autentifican los programas que actúan a nombre de los usuarios. En este sentido, delegar los derechos de un usuario a los programas es una función importante que necesita estar soportada en la capa de conectividad. Regresaremos a la delegación de manera extensiva cuando expliquemos la seguridad aplicada en los sistemas distribuidos.

La *capa de recursos* es responsable de la administración de un solo recurso. Utiliza las funciones proporcionadas por la capa de conectividad y llama de manera directa a las interfaces puestas a disposición mediante la capa de fabricación. Por ejemplo, esta capa ofrece funciones útiles para obtener la información de configuración sobre un recurso específico, o, en general, para realizar operaciones específicas tales como la creación de un proceso o la lectura de datos. De esta manera, la capa de recursos aparece como responsable del control de acceso, y por tanto se apoyará en la autentificación realizada como parte de la capa de conectividad.

La siguiente capa en la jerarquía es la *capa colectiva*. Se encarga de manipular el acceso a múltiples recursos y, por lo general, consta de servicios para descubrir recursos, ubicación y calendarización de tareas dentro de múltiples recursos, replicación de datos, y así sucesivamente. A diferencia de las capas de conectividad y de recursos, que constan de una colección estándar de protocolos relativamente pequeña, la capa colectiva pudiera consistir en muchos protocolos diferentes para muchos propósitos diferentes, reflejando el amplio espectro de servicios que puede ofrecer a una organización virtual.

Por último, la *capa de aplicaciones* consta de aplicaciones que operan dentro de una organización virtual y hacen uso del ambiente de cómputo en grid.

Por lo general, las capas colectiva, de conectividad, y de recursos forman el núcleo de lo que podríamos llamar una capa grid middleware. Juntas, estas capas proporcionan acceso y administración para los recursos que están potencialmente dispersos a través de muchos sitios. Una observación importante, desde la perspectiva del middleware, es que con la computación en grid la idea de un sitio (o unidad de administración) es común. Este predominio se enfatiza mediante el intercambio gradual hacia una **arquitectura orientada al servicio** en la cual los sitios ofrecen acceso a las distintas capas a través del conjunto de servicios de la web (Joseph y cols., 2004). Por ahora, esto ha originado la definición de una arquitectura alternativa conocida como **arquitectura abierta de servicios en grid (OGSA**, por sus siglas en inglés**)**. Esta arquitectura consta de distintos tipos de capas y de muchos componentes, ello la vuelve compleja. La complejidad parece ser la base de cualquier proceso de estandarización. Podemos encontrar detalles sobre la OGSA en Foster y colaboradores (2005).

1.3.2 Sistemas distribuidos de información

Podemos encontrar otra importante clase de sistemas distribuidos en organizaciones que fueron confrontadas con aplicaciones de red saludables, pero cuya interoperatividad se volvió una experiencia amarga. Muchas de las soluciones middleware existentes son resultado del trabajo con la infraestructura en la cual era más fácil integrar aplicaciones dentro de un sistema de información corporativa (Bernstein, 1996; y Alonso y cols., 2004).

Podemos distinguir varios niveles donde la integración tomó parte. En muchos casos, una aplicación de red constaba simplemente de un servidor que ejecutaba esa aplicación (con frecuencia incluye una base de datos) y la dejaba disponible para programas remotos, llamados **clientes**. Dichos clientes podían enviar una petición al servidor, para ejecutar alguna operación específica, después de la cual se enviaba una respuesta. La integración al nivel más bajo permitía a los clientes alojar cierto número de peticiones, posiblemente para distintos servidores, dentro de una sola petición más grande y ejecutarla como una **transacción distribuida**. La idea clave era que todas, o ninguna de las peticiones, pudieran ser ejecutadas.

Mientras más sofisticadas se hicieron las aplicaciones y se fueron separando de manera gradual en componentes independientes (distinguiendo entre los componentes de base de datos y los componentes de proceso), se volvió claro que la integración también debe tomar parte al dejar que las aplicaciones se comuniquen directamente entre sí. Esto ha derivado en una gran industria que se concentra en la **integración de aplicaciones empresariales** (**EAI**, por sus siglas en inglés). A continuación, nos concentraremos en estas dos formas de sistemas distribuidos.

Sistemas de procesamiento de transacciones

Para clarificar nuestra explicación, concentrémonos en aplicaciones de bases de datos. En la práctica, en una base de datos, las operaciones se llevan a cabo generalmente en forma de **transacciones**. Programar utilizando transacciones requiere primitivas de transacción especiales que deben ser proporcionadas ya sea por el sistema distribuido subyacente o por el lenguaje del sistema en tiempo

de ejecución. Ejemplos típicos de primitivas de transacción aparecen en la figura 1-8. La lista exacta depende de qué tipo de objetos se utilizan en la transacción (Gray y Reuter, 1993). En un sistema de correo, es posible que existan transacciones primitivas para enviar, recibir, y reenviar correos. Sin embargo, en un sistema contable puede ser muy diferente; READ y WRITE son ejemplos clásicos. Instrucciones ordinarias, llamadas a procedimientos, etc., también son permitidas dentro de una instrucción. En particular, decimos que las llamadas a procedimientos remotos (RPC por sus siglas en inglés), es decir, llamadas a procedimientos de servidores remotos, con frecuencia también se encapsulan en una transacción, dando paso a lo que se conoce como **RPC transaccional**. En el capítulo 4 explicaremos ampliamente las RPC (por sus siglas inglés).

Primitiva	Descripción
BEGIN_TRANSACTION	Marca el inicio de una transacción
END_TRANSACTION	Termina la transacción e intenta continuar
ABORT_TRANSACTION	Finaliza la transacción y restablece los viejos valores
READ	Lee los datos desde un archivo una tabla, u otra fuente
WRITE	Escribe los datos en un archivo una tabla, o en otra fuente

Figura 1-8. Ejemplo de primitivas de transacción.

BEGIN_TRANSACTION y END_TRANSACTION se utilizan para delimitar el alcance de una transacción. Las operaciones entre estas primitivas forman el cuerpo de la transacción. La característica clásica de una transacción es que, o se ejecutan todas sus operaciones, o no se ejecuta ninguna operación. Las operaciones pueden ser llamadas del sistema, procedimientos de biblioteca, o instrucciones de un lenguaje encerradas entre paréntesis, según la implementación.

Esta propiedad de todo o nada es una de las cuatro características que tienen las transacciones. De manera más específica, las transacciones son:

1. Atómicas: para el mundo exterior, la transacción es indivisible.
2. Consistentes: la transacción no viola sistemas invariantes.
3. Aisladas: las transacciones concurrentes no interfieren entre sí.
4. Durables: una vez que se confirma una transacción, los cambios son permanentes.

Estas propiedades con frecuencia son referidas mediante sus letras iniciales: **ACAD** (o **ACID**, por sus iniciales en inglés).

La primera propiedad clave que presentan todas las transacciones es que son **atómicas**. Esta propiedad garantiza que cada transacción ocurra completamente, o se omite, y si ocurre, sucede en una sola acción instantánea e indivisible. Mientras una transacción se encuentra en progreso, otros procesos (estén o no involucrados en la transacción) no pueden ver ningún estado intermedio.

La segunda propiedad de las transacciones dice que son **consistentes**. Lo cual significa que si el sistema tiene ciertas invariantes que deben permanecer siempre, si se mantuvieron antes de la transacción, también permanecerán después. Por ejemplo, en un sistema bancario, una invariante

clave es la ley de conservación de dinero. Después de cada transferencia interna, la cantidad de dinero presente en el banco debe ser la misma que existía antes de la transferencia, pero por un breve momento durante la transacción, esta invariante puede violarse. Sin embargo, la violación no es visible fuera de la transacción.

La tercera propiedad dice que las transacciones son **aisladas** o **en serie**. Esto significa que si dos o más transacciones se están ejecutando al mismo tiempo, para cada una de ellas y para otros procesos, el resultado final luce como si todas las transacciones se ejecutaran en secuencia, en cierto orden (según el sistema).

La cuarta propiedad dice que las transacciones son **durables**. Esto se refiere al hecho de que una vez confirmada una transacción, no importa qué suceda, la transacción continúa y los resultados se vuelven permanentes. Ninguna falla ocurrida después de la confirmación puede deshacer los resultados, u ocasionar que se pierdan. (En el capítulo 8 explicaremos ampliamente la durabilidad.)

Hasta ahora, hemos definido las transacciones en una sola base de datos. Una **transacción anidada** se construye a partir de cierta cantidad de subtransacciones, como indica la figura 1-9. La transacción de más alto nivel puede dividirse en subprocesos hijos que se ejecutan en paralelo entre sí, en diferentes máquinas, para mejorar el rendimiento o simplificar la programación. Cada uno de estos subprocesos también puede ejecutar una o más subtransacciones, o dividirse en sus propios subprocesos hijos.

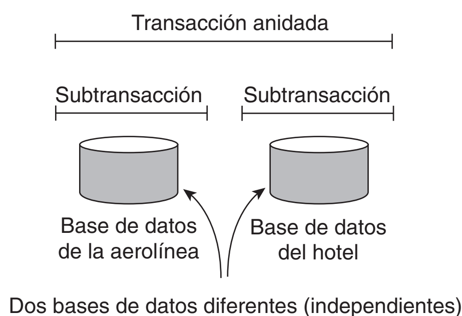


Figura 1-9. Transacción anidada.

Las subtransacciones dan lugar a un sutil, pero importante, problema. Imagine que una transacción inicia diversas subtransacciones en paralelo, y que una de éstas se confirma volviendo visibles los resultados para la transacción padre. Después de más cálculos, la transacción padre aborta y restablece todo el sistema en el estado que tenía antes de iniciada la transacción de más alto nivel. Como consecuencia, los resultados de la subtransacción que se confirmó, a pesar de todo, deben deshacerse. Por tanto, la permanencia a la que nos referimos antes sólo es aplicable a transacciones del más alto nivel.

Debido a que las transacciones pueden anidarse muy profundamente de modo arbitrario, se necesita una buena administración para lograr que todo salga bien. Sin embargo, la semántica es clara. Cuando usted inicia cualquier transacción o subtransacción, en el sistema se proporciona conceptualmente una copia privada de todos los datos para que la manipule como desee; si aborta, su universo privado simplemente desaparece, como si nunca hubiera existido. Si se confirma, su universo privado reemplaza al universo padre. Por tanto, si una subtransacción se confirma, y después

se inicia una subtransacción nueva, la segunda ve el resultado producido por la primera. De igual modo, si una transacción envolvente (de más alto nivel) aborta, todas sus subtransacciones también deben abortarse.

En los sistemas distribuidos, las transacciones anidadas son importantes para que proporcionen una forma natural de distribuir una transacción a través de varias máquinas. Las transacciones anidadas siguen una división *lógica* del trabajo de la transacción original. Por ejemplo, una transacción para planificar un viaje para el que es necesario reservar tres diferentes vuelos, puede dividirse de manera lógica en tres subtransacciones. Cada una de estas subtransacciones puede administrarse separada e independientemente de las otras dos.

En los primeros días de los sistemas middleware empresariales, el componente que manejaba transacciones distribuidas (o anidadas) conformaba la parte central para integrar aplicaciones al nivel de servidor o de base de datos. A este componente se le llamó **monitor de procesamiento de transacciones** o **monitor TP**, para abreviar. Su tarea principal era permitirle a una aplicación el acceso a múltiples servidores/bases de datos ofreciendo un modelo de programación transaccional, como indica la figura 1-10.

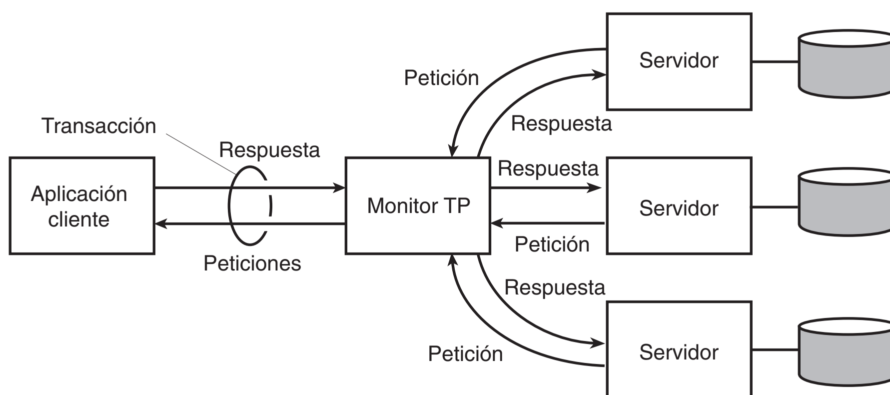


Figura 1-10. Rol de un monitor TP en sistemas distribuidos.

Integración de aplicaciones empresariales

Como ya mencionamos, conforme las aplicaciones fueron desligándose de las bases de datos sobre las que se construyeron, la necesidad de que las instalaciones integraran aplicaciones independientes de sus bases de datos se volvió más evidente. En particular, los componentes de las aplicaciones debían ser capaces de comunicarse entre sí de manera directa y no sólo mediante un comportamiento de petición-respuesta soportado por sistemas de procesamiento de transacciones.

Esta necesidad de comunicación entre aplicaciones dio pie a muchos modelos de comunicación diferentes, los cuales explicaremos con detalle en este libro (y por ello, debemos abreviar de momento). La idea principal fue que aplicaciones existentes pudieran intercambiar información de manera directa, como ilustra la figura 1-11.

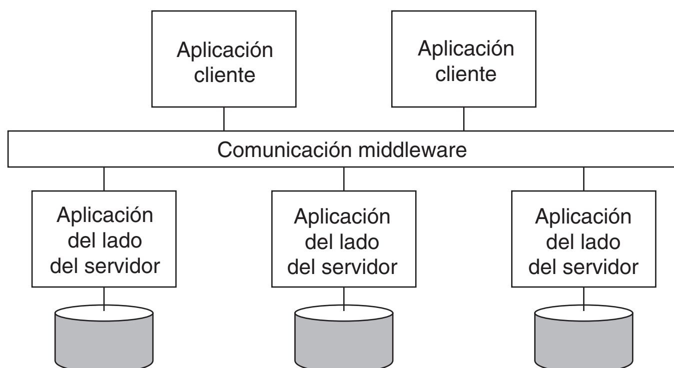


Figura 1-11. Middleware como facilitador de la comunicación al integrar aplicaciones empresariales.

Existen diversos tipos de comunicación middleware. Con las **llamadas a procedimientos remotos** (**RPC**), un componente de aplicación puede enviar de manera efectiva una petición a otro componente de aplicación, si realiza una llamada a un procedimiento local, lo cual resulta en una petición que se empaca como un mensaje y se envía al componente invocado (remoto). De igual manera, el resultado se enviará de regreso y será devuelto a la aplicación como resultado de la llamada al procedimiento.

Conforme aumentó la popularidad de la tecnología de objetos se desarrollaron técnicas para permitir llamadas a objetos remotos, ello dio pie a lo que se conoce como **invocaciones a métodos remotos** (**RMI**, por sus siglas en inglés). Una RMI es básicamente lo mismo que una RPC, excepto que la RMI opera sobre objetos en lugar de aplicaciones.

RPC y RMI tienen la desventaja de que el que llama y el llamado necesitan estar en ejecución y estar activos al momento de la comunicación. Además, necesitan saber exactamente cómo referirse uno al otro. Esta estrecha relación con frecuencia representa un serio inconveniente, y ha dado lugar a lo que se conoce como **middleware orientado a mensajes**, o **MOM**. En este caso, las aplicaciones simplemente envían mensajes a puntos de contacto lógicos, en general descritos mediante un sujeto. De igual manera, las aplicaciones pueden indicar su interés por un tipo específico de mensaje, después de lo cual la comunicación middleware se encargará de que esos mensajes se entreguen a aquellas aplicaciones. Estos sistemas, también conocidos como sistemas de **publicación-suscripción**, forman una clase importante y creciente de sistemas distribuidos, la cual explicaremos en el capítulo 13.

1.3.3 Sistemas distribuidos masivos

Los sistemas distribuidos que hemos explicado hasta el momento se caracterizan principalmente por su estabilidad: los nodos están fijos y tienen una conexión más o menos permanente hacia una red. Hasta cierto punto, esta estabilidad se logra a través de las diversas técnicas que explicamos en este libro, y las cuales intentan lograr cierta transparencia en la distribución. Por ejemplo, la rique-

za de las técnicas para enmascarar fallas, y recuperarse de ellas, dará la impresión de que sólo ocasionalmente las cosas salen mal. De igual manera, hemos podido ocultar aspectos relacionados con la ubicación real de un nodo al permitir de manera efectiva que usuarios y aplicaciones crean que los nodos permanecen fijos.

Sin embargo, la situación se ha vuelto muy distinta con la introducción de dispositivos de cómputo móviles y embebidos. Ahora nos enfrentamos con sistemas distribuidos en los cuales la inestabilidad es el comportamiento predeterminado. En estos sistemas, a los que nos referimos como **sistemas distribuidos masivos**, con frecuencia los dispositivos se caracterizan por ser pequeños, de baterías, portátiles, y tienen sólo una conexión inalámbrica, aunque no todas estas características son aplicables a todos los dispositivos. Más aún, tales atributos no deben interpretarse como necesariamente restrictivos, como lo demuestran todas las posibilidades de los modernos teléfonos inteligentes (Roussos y cols., 2005).

Tal como sugiere su nombre, un sistema distribuido móvil es parte de nuestro entorno (y como tal, está inherentemente distribuido). Una característica importante es su carencia general de control administrativo humano. En el mejor de los casos, los dispositivos son configurados por sus propietarios, ya que de otro modo necesitan descubrir automáticamente su ambiente y "adaptarse" de la mejor manera posible. Grimm y colaboradores (2004) precisaron bastante esta adaptación al formular los tres siguientes requerimientos para aplicaciones móviles:

1. Incluir cambios contextuales.
2. Fomentar composiciones a la medida.
3. Reconocer el intercambio como algo común.

Incluir cambios contextuales significa que un dispositivo debe estar continuamente consciente de que su ambiente puede cambiar en cualquier momento. Uno de los cambios más sencillos es descubrir que una red ya no está disponible, por ejemplo, debido a que un usuario se está moviendo entre estaciones base. En tales casos, la aplicación debe reaccionar, probablemente conectándose automáticamente a otra red, o tomando otras acciones adecuadas.

Fomentar composiciones a la medida se refiere al hecho de que muchos dispositivos de sistemas masivos se utilizarán en formas muy diferentes por distintos usuarios. Como resultado, la suite de aplicaciones que se ejecutan en un dispositivo debe ser sencilla de configurar, ya sea por el usuario o mediante la interpolación automatizada (pero controlada).

Un aspecto muy importante de los sistemas dominantes es que los dispositivos generalmente ingresan al sistema para acceder a información (y probablemente proporcionarla). Esto sucede para facilitar la lectura, el almacenamiento, e intercambiar (compartir) información. Debido a la intermitente y cambiante conectividad de los dispositivos, el espacio donde reside la información accesible muy probablemente cambiará a cada momento.

Mascolo y colaboradores (2004), así como Niemela y Latvakoski (2004), llegaron a conclusiones similares: ante la movilidad, los dispositivos deben soportar una sencilla y dependiente adaptación a su ambiente local. Deben ser capaces de descubrir eficientemente servicios, y de reaccionar en consecuencia. Debe quedar claro, a partir de estos requerimientos, que la transparencia en la

distribución en realidad no sucede en los sistemas masivos. De hecho, la distribución de datos, procesos, y control es *inherente* a estos sistemas, razón por la cual puede ser mejor simplemente exponerla en lugar de intentar ocultarla. Ahora veamos algunos ejemplos concretos sobre sistemas masivos.

Sistemas caseros

Un tipo de sistema masivo cada vez más popular, pero que tal vez sea el menos restringido, es el sistema construido alrededor de una red casera. Estos sistemas generalmente consisten en una o más computadoras personales, aunque lo más importante es que integran aparatos electrónicos de consumo típicos como televisiones, equipos de audio y video, dispositivos para juegos electrónicos, teléfonos (inteligentes), PDA, y otros aparatos personales en un solo sistema. Además, podemos esperar que todo tipo de dispositivos tales como implementos de cocina, cámaras de vigilancia, relojes, controladores de alumbrado, etc., estarán conectados a un solo sistema distribuido.

Desde una perspectiva de sistemas, existen diversos retos a vencer antes de que los sistemas caseros masivos se vuelvan una realidad. Un desafío importante es que un sistema debe ser completamente autoconfigurable y autoadministrable. No se puede esperar que los usuarios finales estén dispuestos a, y sean capaces de, mantener en ejecución un sistema distribuido casero si sus componentes son propensos a errores (como es el caso de muchos dispositivos actuales). Mucho se ha logrado ya con los estándares del **Plug and Play Universal (UPnP)**, por medio del cual los dispositivos automáticamente obtienen direcciones IP, pueden localizarse uno a otro, etc. (Foro UPnP, 2003), pero se necesita más. Por ejemplo, no queda claro cómo es que en los dispositivos tanto el software como el firmware puedan actualizarse fácilmente sin intervención manual, o cuándo deben ocurrir las actualizaciones de tal manera que no se viole la compatibilidad con otros dispositivos.

Otro motivo de tensión es la administración de lo que se conoce como "espacio personal". Al reconocer que un sistema casero consta de muchos dispositivos compartidos, así como personales, y que en un sistema casero los datos también están sujetos a restricciones de intercambio, se necesita de mucha atención para realizar tales espacios personales. Por ejemplo, parte del espacio personal de Alice puede consistir en su agenda, sus fotos familiares, un diario, música y videos que compró, etc. Estos artículos personales deben almacenarse de tal forma que Alice tenga acceso a ellos siempre que lo desee. Más aún, parte de este espacio personal debe ser (temporalmente) accesible para otros, por ejemplo, cuando necesite hacer una cita de negocios.

Por fortuna, las cosas pueden volverse más sencillas. Durante mucho tiempo se pensó que los espacios personales relacionados con sistemas caseros estaban inherentemente distribuidos mediante diversos dispositivos. Desde luego, tal dispersión puede conducirnos fácilmente a importantes problemas de sincronización. Sin embargo, los problemas pueden resolverse debido al rápido incremento en la capacidad de los discos duros, junto con su disminución en tamaño. Configurar una unidad de almacenamiento multiterabyte para una computadora personal en realidad no es un problema. Al mismo tiempo, los discos duros portátiles que tienen capacidades de cientos de gigabytes se han colocado dentro de reproductores de medios portátiles relativamente pequeños. Con estas capacidades que continuamente aumentan, podemos ver sistemas caseros dominantes que

adoptan una arquitectura en la que una sola máquina actúa como un maestro (y está oculta en alguna parte del sótano, cerca de la calefacción central), y todos los demás dispositivos fijos simplemente proporcionan una interfaz conveniente para los seres humanos. Los dispositivos personales estarán entonces abarrotados con información diaria necesaria, pero nunca les faltará capacidad.

Sin embargo, tener suficiente capacidad de almacenamiento no resuelve el problema de administrar espacios personales. Ser capaz de almacenar enormes cantidades de datos cambia el problema a almacenar datos *relevantes* y poder encontrarlos después. Con mayor frecuencia veremos sistemas masivos, tales como redes caseras, equipados con lo que se conoce como **asesores**, programas que consultan lo que otros usuarios han almacenado para identificar gustos similares, y a partir de ahí deducir qué contenido colocar en el espacio personal de cada quien. Una observación interesante es la cantidad de información que los programas asesores necesitan para hacer su trabajo, la cual es con frecuencia lo suficientemente pequeña como para permitir que se ejecuten en una PDA (Miller y cols., 2004).

Sistemas electrónicos para el cuidado de la salud

Otra clase importante y reciente de sistemas masivos es la que comprende los sistemas relacionados con dispositivos electrónicos (personales) utilizados para el cuidado de la salud. Con el creciente costo de los tratamientos médicos, se están desarrollando nuevos dispositivos para dar seguimiento al bienestar de las personas con el fin de que se pongan en contacto automáticamente con los médicos cuando lo necesiten. En muchos de estos sistemas, el principal objetivo es evitar que la gente sea hospitalizada.

Los sistemas para el cuidado de la salud con frecuencia están equipados con diversos sensores organizados en una BAN (*body-area network*; red de área corporal), de preferencia inalámbrica. Un asunto importante es que dicha red sólo debe obstaculizar al mínimo a una persona. Para lograr este fin, la red debe poder operar mientras la persona está en movimiento, sin restricciones (es decir, cables) relacionadas con dispositivos fijos.

Este requerimiento deriva en dos organizaciones evidentes, como ilustra la figura 1-12. En la primera, un *hub* central es parte de la BAN, y reúne la información necesaria. De tiempo en tiempo, esta información se libera en un dispositivo de almacenamiento más grande. La ventaja de este esquema es que el hub también puede controlar la BAN. En el segundo escenario, la BAN está conectada de modo continuo a una red externa —de nuevo mediante una conexión inalámbrica— a la cual se envía la información monitoreada. Para administrar la BAN se necesitarán técnicas separadas. Por supuesto, también pueden existir conexiones adicionales con el médico u otras personas.

Desde una perspectiva de sistemas distribuidos, de inmediato nos enfrentamos con preguntas como:

1. ¿En dónde y cómo debe almacenarse la información monitoreada?

2. ¿Cómo puede evitarse la pérdida de información crucial?

3. ¿Qué infraestructura se necesita para generar y propagar alertas?

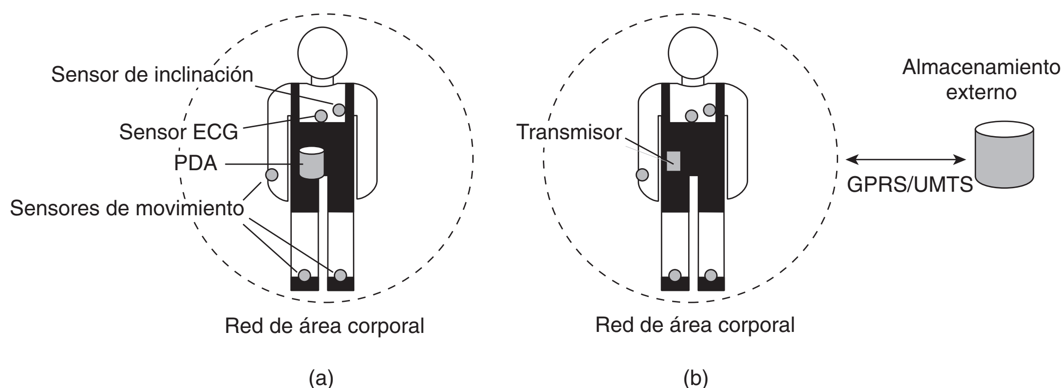


Figura 1-12. Monitoreo de una persona mediante un sistema electrónico dominante para el cuidado de la salud utilizando (a) un hub local, o (b) una conexión inalámbrica continua.

4. ¿Cómo pueden proporcionar los médicos retroalimentación en línea?

5. ¿Cómo puede lograrse una fortaleza extrema en el sistema de monitoreo?

6. ¿Cuáles son los temas de seguridad y cómo se pueden reforzar las políticas adecuadas?

A diferencia de los sistemas caseros, no podemos esperar que la arquitectura de los sistemas dominantes para el cuidado de la salud se muevan hacia sistemas de un solo servidor y hacer que los dispositivos de monitoreo operen con un mínimo de funcionalidad. Al contrario, por razones de eficiencia: tanto dispositivos como redes de área corporal serán requeridos para dar soporte al **procesamiento de datos dentro de la red**, lo cual significa que al monitorear la información, ésta deberá, por ejemplo, agregarse, almacenarse, o enviarse permanentemente a un médico. A diferencia del caso de los sistemas distribuidos de información, aún no tenemos respuestas claras para estas preguntas.

Redes de monitoreo

Nuestro último ejemplo de sistemas masivos es el de redes de monitoreo. En muchos casos, estas redes forman parte de la tecnología que permite el dominio, y vemos que muchas soluciones para las redes de monitoreo se vuelven aplicaciones masivas. Lo que hace interesantes a las redes de monitoreo, desde una perspectiva de sistemas distribuidos, es que en prácticamente todos los casos se utilizan para procesar información. En este sentido, hacen más que sólo proporcionar servicios de comunicación, para lo cual tradicionalmente sirven las redes de computadoras. Akyildiz y colaboradores (2002) proporciona un panorama desde una perspectiva de redes. Zhao y Guibas (2004) presentan una introducción más orientada a sistemas de redes de monitoreo. Otras redes fuertemente relacionadas son las **redes mesh**, las cuales básicamente forman una colección de nodos (fijos) que se comunican a través de vínculos inalámbricos. Estas redes pueden formar la base para muchos sistemas distribuidos de media escala; Akyildiz y colaboradores (2005) proporcionan un panorama a este respecto.

Una red de sensores típica está constituida por decenas, centenas o miles de nodos relativamente pequeños, y cada nodo está equipado con un dispositivo sensor. La mayoría de las redes de monitoreo utiliza la comunicación inalámbrica, y los nodos generalmente son de baterías. Sus limitados recursos, sus restringidas capacidades de comunicación, y su consumo de energía, demandan en la lista de criterios de diseño que la eficiencia sea alta.

La relación con sistemas distribuidos puede aclararse si consideramos a las redes de monitoreo como bases de datos distribuidas. Este enfoque resulta muy común y fácil de entender cuando nos damos cuenta de que las redes de monitoreo se utilizan para aplicaciones de medición y vigilancia (Bonnet y cols., 2002). En estos casos, un operador querría extraer información desde (una parte de) la red haciendo simplemente consultas tales como: ¿Cuál es la carga de tránsito en la carretera 1? Estas consultas se parecen a las de las bases de datos tradicionales. En este caso, es probable que para dar la respuesta se necesite la colaboración de muchos sensores ubicados a lo largo de la carretera 1, mientras que otros sensores quedan intactos.

Para organizar una red de monitoreo como una base de datos distribuida, existen básicamente dos extremos, según muestra la figura 1-13. Primero, los sensores no cooperan sino que simplemente envían sus datos a la base de datos centralizada, ubicada en el lugar del operador. El otro extremo es para enviar consultas a sensores importantes y dejar que cada sensor calcule una respuesta, ello requiere que el operador agregue las respuestas devueltas.

Ninguna de estas soluciones es muy atractiva. La primera requiere que los sensores envíen todos los datos medidos a través de la red, lo cual puede hacer que se desperdicien recursos de la red y energía. La segunda solución también puede provocar el desperdicio de recursos cuando descarta las capacidades de agregación de los sensores, lo cual permitiría que menos datos regresaran al operador. Lo que se necesita son instalaciones para el **procesamiento de datos dentro de la red**, y esto lo encontramos también en sistemas masivos para el cuidado de la salud.

Dentro de la red, el procesamiento puede hacerse de diversas formas. Una manera evidente es enviar una consulta a todos los nodos sensores, a lo largo de un árbol que comprenda todos los nodos, y posteriormente agregar los resultados conforme se propaguen de regreso a la raíz, donde se ubica el iniciador. La agregación tendrá lugar en donde se junten dos o más ramas del árbol. Así de sencillo como parece, este sistema introduce preguntas difíciles:

1. ¿Cómo configuramos (dinámicamente) un árbol eficiente en una red de monitoreo?

2. ¿Cómo se lleva a cabo la agregación de resultados? ¿Es posible controlarla?

3. ¿Qué sucede cuando los vínculos de la red fallan?

Estas preguntas se han solucionado parcialmente en TinyDB, la cual implementa una interfaz declarativa (base de datos) hacia redes de sensores inalámbricos. En esencia, TinyDB puede utilizar cualquier algoritmo para rutinas basadas en árboles. Un nodo intermedio recopilará y agregará los resultados de sus hijos, junto con sus propios resultados, y los enviará hacia la raíz. Para hacer las cosas eficientes, las consultas son espaciadas durante un periodo que permite programar cuidadosamente las operaciones de tal manera que los recursos de la red y la energía se consuman en forma óptima. Pueden encontrarse los detalles en Madden y cols. (2005).

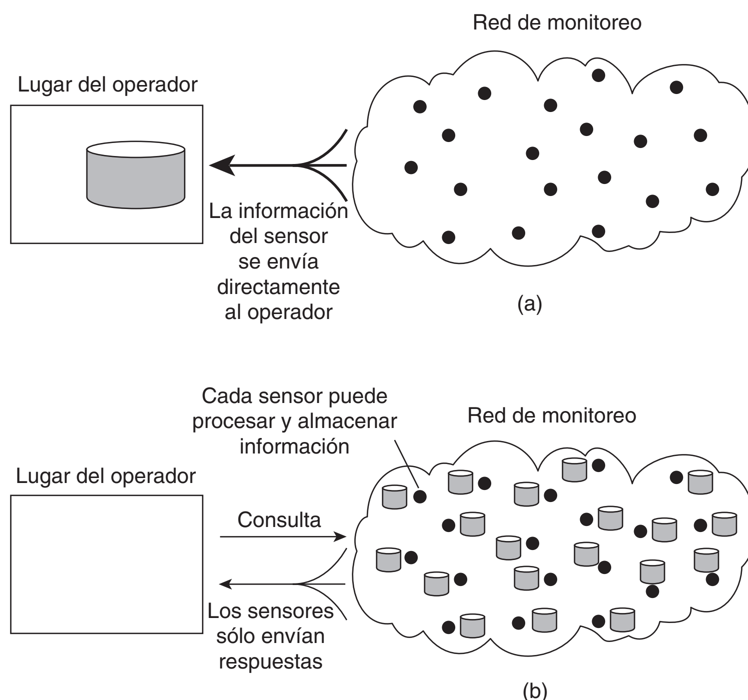


Figura 1-13. Organización de una base de datos para una red de monitoreo, mientras se almacena y procesa información (a) sólo en el lugar del operador, o (b) sólo en los sensores.

Sin embargo, como las consultas pueden iniciarse desde diferentes puntos de la red, utilizar árboles de una sola raíz, como en TinyDB, puede no ser lo suficientemente eficiente. Como alternativa, las redes de sensores pueden equiparse con nodos especiales donde los resultados se reenvían, así como las consultas relacionadas con esos resultados. A modo de simple ejemplo, consultas y resultados que se relacionen con lecturas de temperatura se recopilan en diferentes ubicaciones, a diferencia de aquellos relacionados con mediciones de humedad. Este método corresponde directamente a la idea de los sistemas publicación-suscripción, los cuales explicaremos ampliamente en el capítulo 13.

1.4 RESUMEN

Los sistemas distribuidos consisten en computadoras autónomas que trabajan juntas para dar la apariencia de un solo sistema coherente. Una ventaja importante es que facilitan la integración de diferentes aplicaciones que se ejecutan en distintas computadoras conectadas en un solo sistema. Otra ventaja es que cuando se diseñan adecuadamente, los sistemas distribuidos se escalan muy

bien con respecto al tamaño de la red subyacente. Estas ventajas con frecuencia tienen que pagar el precio de un software más complejo, la degradación del rendimiento, y también debilitan la seguridad. Sin embargo, a nivel mundial existe un gran interés en la construcción e instalación de sistemas distribuidos.

Los sistemas distribuidos con frecuencia intentan ocultar mucho de lo intrincado de la distribución de procesos, datos, y control. Sin embargo, esta transparencia en la distribución no sólo implica un costo en el rendimiento, sino en situaciones prácticas que nunca se logran por completo. El hecho de que sea necesario compensar para lograr varias formas de transparencia en la distribución es inherente al diseño de sistemas distribuidos, y puede fácilmente complicar su entendimiento.

Las cosas se complican más por el hecho de que muchos desarrolladores inicialmente hacen muchas suposiciones sobre la red subyacente, las cuales son básicamente erróneas. Después, cuando las suposiciones se eliminan, puede resultar difícil enmascarar un comportamiento no deseado. Un ejemplo típico es suponer que la latencia de la red no es importante. Posteriormente, cuando se transporta un sistema existente a una red de área amplia, ocultar las latencias puede afectar profundamente el diseño original del sistema. Otros escollos incluyen suponer que la red es confiable, estática, segura, y homogénea.

Existen diferentes tipos de sistemas distribuidos, los cuales pueden clasificarse como orientados hacia el soporte de cálculos, procesamiento de información, y masivos. Los sistemas de cómputo distribuidos generalmente se utilizan para aplicaciones de alto rendimiento que se originan de modo habitual en el campo del cómputo en paralelo. Una amplia clase de sistemas distribuidos puede encontrarse en los ambientes tradicionales de oficina, donde vemos a las bases de datos representando un papel importante. Por lo general, los sistemas de procesamiento de transacciones se utilizan en estos ambientes. Por último, una clase de sistemas distribuidos de reciente aparición es aquella donde los componentes son pequeños y el sistema se forma a la medida, pero que sobre todo no es manejado por un administrador de sistemas. Esta última clase generalmente es representada por ambientes de cómputo ubicuos.

PROBLEMAS

1. Una definición alternativa para un sistema distribuido es que una colección de computadoras independientes dé la apariencia de ser un solo sistema, es decir, que esté completamente oculto para los usuarios que se encuentren incluso en diferentes computadoras. Proporcione un ejemplo en el que este enfoque sea muy evidente.
2. ¿Cuál es el papel del middleware en un sistema distribuido?
3. Muchos sistemas en red están organizados en términos del back office y el front office. ¿Cómo es que las empresas coinciden con el punto de vista coherente que demandamos para un sistema distribuido?
4. Explique lo que intentamos decir con transparencia (de distribución), y proporcione ejemplos de distintos tipos de transparencia.

5. ¿Por qué a veces es tan difícil ocultar la ocurrencia y la recuperación de fallas en un sistema distribuido?
6. ¿Por qué no siempre es buena idea intentar implementar el grado más alto de transparencia?
7. ¿Qué es un sistema distribuido abierto, y qué beneficios proporciona la apertura?
8. Describa precisamente lo que significa un sistema escalable.
9. La escalabilidad puede lograrse aplicando distintas técnicas. ¿Cuáles son esas técnicas?
10. Explique lo que quiere decir organización virtual, y proporcione un indicio sobre cómo es que tales organizaciones podrían implementarse.
11. Cuando se aborta una transacción, hemos dicho que el mundo se restaura a su estado anterior, como si la transacción nunca hubiera ocurrido. Mentimos. Proporcione un ejemplo en donde reiniciar el mundo resulte imposible.
12. Ejecutar transacciones anidadas requiere cierta coordinación. Explique lo que debe hacer en realidad un coordinador.
13. Argumentamos que la transparencia de distribución puede no ocurrir en sistemas masivos. Esta aseveración no es verdadera para todos los tipos de transparencia. Proporcione un ejemplo.
14. Nosotros ya le dimos algunos ejemplos de sistemas masivos distribuidos: sistemas caseros, sistemas electrónicos para el cuidado de la salud, y redes de monitoreo. Amplíe usted esta lista con más ejemplos.
15. **(Asignación para el laboratorio.)** Esboce un diseño para un sistema casero consistente en un servidor de medios que permita la conexión de un cliente inalámbrico. Este último se conecta a un equipo (analógico) de audio-video y transforma los flujos de medios digitales en una salida analógica. El servidor se ejecuta en una máquina por separado, posiblemente conectada a internet, pero no tiene un teclado o ningún monitor conectados.

2

ARQUITECTURAS

Por lo general, los sistemas distribuidos son complejas piezas de software cuyos componentes se encuentran, por definición, dispersos en diversas máquinas. Para dominar esta complejidad, resulta crucial que los sistemas se encuentren organizados adecuadamente. Existen diferentes formas de visualizar la organización de un sistema distribuido, pero un modo evidente es saber diferenciar la organización lógica de la colección de componentes del software de la organización física real.

La organización de los sistemas distribuidos trata básicamente sobre los componentes de software que constituyen el sistema. Estas **arquitecturas de software** nos dicen cómo se organizarán los componentes de software, y cómo deben interactuar. En este capítulo veremos primero algunos métodos aplicados comúnmente para organizar sistemas (distribuidos) de cómputo.

La organización real de un sistema distribuido requiere que generemos las instancias y coloquemos los componentes del software en máquinas reales. Existen diferentes alternativas para hacer esto. La creación de instancias finales de una arquitectura de software también se conoce como **arquitectura de sistema**. En este capítulo veremos arquitecturas centralizadas tradicionales en las que un solo servidor implementa la mayoría de los componentes de software (y, por tanto, la funcionalidad), mientras que los clientes remotos pueden acceder a ese servidor utilizando medios de comunicación simples. Además, consideraremos arquitecturas descentralizadas en las que las máquinas desempeñan roles casi iguales, así como organizaciones híbridas.

Tal como explicamos en el capítulo 1, un objetivo importante de los sistemas distribuidos es separar las aplicaciones de las plataformas subyacentes mediante una capa middleware. Adoptar tal capa es una decisión arquitectónica importante, y su objetivo principal es proporcionar

transparencia de distribución. Sin embargo, es necesario negociar para lograr la transparencia, lo cual nos lleva a implementar diversas técnicas para adaptar el middleware; en este capítulo explicaremos algunas de las técnicas más aplicadas y cómo afectan la organización del propio middleware.

En los sistemas distribuidos, la adaptabilidad también puede lograrse haciendo que el sistema monitoree su propio comportamiento y tome las medidas adecuadas cuando sea necesario. Esto ha dado pie a una clase de lo que ahora conocemos como **sistemas autónomos**. Estos sistemas distribuidos frecuentemente se organizan en forma de ciclos de control de retroalimentación, los cuales forman un elemento arquitectónico importante durante el diseño de un sistema. En este capítulo dedicamos una sección a los sistemas distribuidos autónomos.

2.1 ESTILOS (MODELOS) ARQUITECTÓNICOS

Iniciamos nuestra explicación sobre arquitecturas considerando primero la organización lógica de los sistemas distribuidos en componentes de software, también conocida como arquitectura de software (Bass y cols., 2003). La investigación sobre arquitecturas de software ha madurado considerablemente, y ahora es común aceptar que el diseño o la adopción de una arquitectura resulta crucial para el desarrollo exitoso de sistemas grandes.

Para nuestro estudio, la idea de **estilo arquitectónico** es importante. Tal estilo se formula en términos de componentes, la forma en que los componentes interactúan entre sí, el intercambio de datos entre los componentes y, por último, en cómo es que estos elementos se configuran juntos en un sistema. Un **componente** es una unidad modular con las interfaces requeridas bien definidas; dicha unidad es reemplazable dentro de su ambiente (OMG, 2004b). Tal como explicaremos más adelante, la cuestión importante sobre un componente para sistemas distribuidos es que pueda ser reemplazado, a condición de respetar sus interfaces. Un concepto de cierta manera más difícil de entender es el de **conector**, el cual generalmente se describe como un mecanismo que media la comunicación, coordinación o cooperación entre componentes (Mehta y cols., 2000; y Shaw y Clements, 1997). Por ejemplo, un conector puede formarse por los medios disponibles para hacer llamadas a procedimientos (remotos), paso de mensajes, o flujo de datos.

Por medio de componentes y conectores podemos lograr varias configuraciones, las cuales se han clasificado en estilos arquitectónicos. Varios estilos ya están identificados y los más importantes para sistemas distribuidos son:

1. Arquitecturas en capas.
2. Arquitecturas basadas en objetos.
3. Arquitecturas centradas en datos.
4. Arquitecturas basadas en eventos.

La idea básica para el estilo en capas es sencilla: los componentes se estructuran (organizan) a **modo de capas**, donde al componente de la capa L_i se le permite llamar a componentes de la capa

subyacente L_{i-1}, pero no del resto de capas, como ilustra la figura 2-1(a). Este estilo se ha adoptado ampliamente en la comunidad de redes, y lo revisaremos brevemente en el capítulo 4. Una observación clave es que el control generalmente fluye de capa a capa: las peticiones se mueven hacia abajo en la jerarquía mientras que los resultados se mueven hacia arriba.

Una organización bastante libre es la que siguen las **arquitecturas basadas en objetos**, la cual aparece en la figura 2-1(b). En esencia, cada objeto corresponde a lo que hemos definido como componente, y estos componentes se conectan a través de un mecanismo de llamadas a procedimientos (remotos). Sin ser sorprendente, esta arquitectura de software coincide con la arquitectura de sistemas cliente-servidor que describimos antes. La arquitectura basada en capas y la basada en objetos aún son los estilos más importantes utilizados para implementar grandes sistemas de software (Bass y cols., 2003).

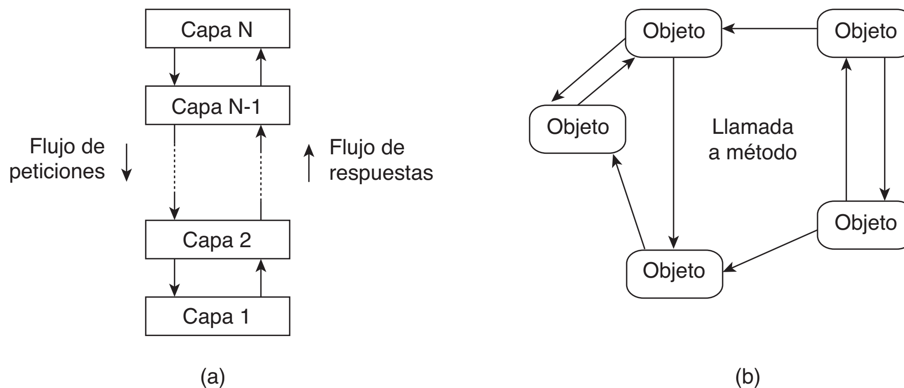


Figura 2-1. Estilos arquitectónicos (a) en capas, y (b) basado en objetos.

Las **arquitecturas centradas en datos** evolucionaron alrededor de la idea de que los procesos se comunican a través de un repositorio común (activo o pasivo). Se puede argumentar que, para sistemas distribuidos, estas arquitecturas son tan importantes como las basadas en capas y objetos. Por ejemplo, un punto a favor que han desarrollado las aplicaciones en red es que se basan en un sistema de archivos distribuidos compartidos donde casi todas las comunicaciones se realizan a través de archivos. De manera similar, los sistemas distribuidos basados en la web, que explicaremos ampliamente en el capítulo 12, se centran bastante en datos: los procesos se comunican a través de servicios de datos compartidos basados en la web.

En las **arquitecturas basadas en eventos**, los procesos se comunican básicamente a través de la propagación de eventos, los que opcionalmente transportan datos, como ilustra la figura 2-2(a). Para sistemas distribuidos, la propagación de eventos se ha asociado con lo que se conoce como **sistemas de publicación-suscripción** (Eugster y cols., 2003). La idea básica es que los procesos publican eventos después de los cuales el middleware asegura que sólo aquellos procesos suscritos a tales eventos los recibirán. La principal ventaja de los sistemas basados en eventos es que los procesos están libremente acoplados. En principio, no necesitan referirse uno a otro explícitamente. A esto se le conoce también como desacoplado en el espacio, o **referencialmente desacoplado**.

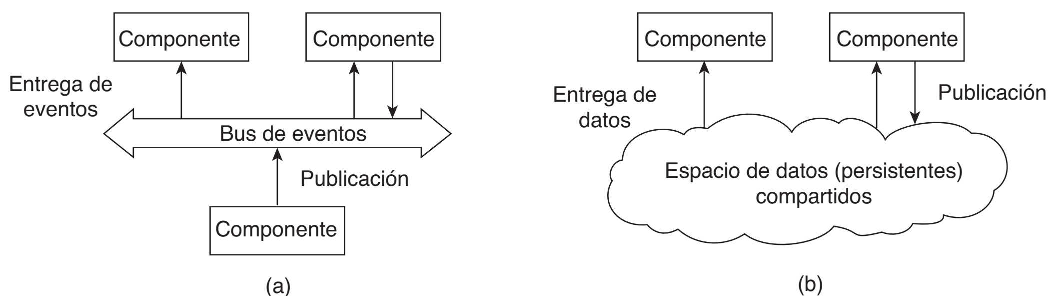


Figura 2-2. Estilo arquitectónico (a) basado en eventos, y (b) espacio de datos compartidos.

Las arquitecturas basadas en eventos pueden combinarse con arquitecturas centradas en datos, y arrojan lo que conocemos como **espacios de datos compartidos**. La esencia de los espacios de datos compartidos es que los procesos ahora también están desacoplados en el tiempo: no es necesario que ambos estén activos cuando la comunicación se lleva a cabo. Además, muchos espacios de datos compartidos utilizan una interfaz similar a la SQL para el repositorio compartido en el sentido de que es posible acceder a los datos utilizando una descripción, en lugar de una referencia explícita, como en el caso de los archivos. Dedicamos el capítulo 13 a este estilo arquitectónico.

Lo que hace importantes a estas arquitecturas de software para los sistemas distribuidos es que todas intentan lograr (a un nivel razonable) la transparencia de distribución. Sin embargo, como hemos explicado, la transparencia de distribución requiere negociar entre el rendimiento, la tolerancia a fallas, la facilidad de programación, etc. Como no existe una solución única que satisfaga todos los requerimientos para todas las aplicaciones distribuidas, los investigadores han abandonado la idea de que un solo sistema distribuido pueda utilizarse para cubrir el 90 por ciento de todos los casos posibles.

2.2 ARQUITECTURAS DE SISTEMAS

Ahora que hemos explicado brevemente algunos estilos arquitectónicos comunes, veamos cuántos sistemas distribuidos están realmente organizados considerando el lugar en donde se colocan los componentes de software. Decidir sobre los componentes de software, sobre su interacción y ubicación, da pie a una instancia de arquitectura de software, también conocida como **arquitectura de sistema** (Bass y cols., 2003). Analizaremos las organizaciones centralizadas y descentralizadas, así como diversas formas híbridas.

2.2.1 Arquitecturas centralizadas

A pesar de que no existe un consenso sobre muchas cuestiones de los sistemas distribuidos, hay un aspecto en el que muchos investigadores y usuarios sí están de acuerdo: pensar en términos de *clientes* que requieren servicios de los *servidores* nos ayuda a comprender y manejar la complejidad de los sistemas distribuidos, y también que son algo bueno.

En el modelo básico cliente-servidor, los procesos de un sistema distribuido se dividen en dos grupos (probablemente traslapados). Un **servidor** es un proceso que implementa un servicio específico, por ejemplo, un servicio de sistema de archivos o un servicio de base de datos. Un **cliente** es un proceso que solicita un servicio a un servidor, enviándole una petición y esperando posteriormente la respuesta. Esta interacción cliente-servidor, también conocida como **comportamiento solicitud-respuesta**, aparece en la figura 2-3.

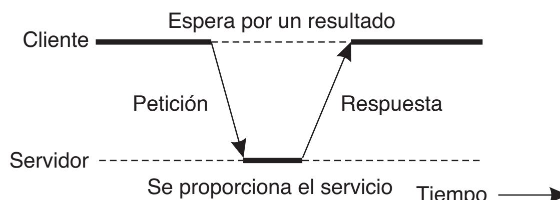


Figura 2-3. Interacción general entre un cliente y un servidor.

La comunicación entre un cliente y un servidor puede implementarse mediante un protocolo simple no orientado a conexión cuando la red subyacente es muy confiable, como sucede en muchas redes de área local. En estos casos, cuando un cliente solicita un servicio, simplemente empaca un mensaje para el servidor, identificando el servicio que requiere, junto con la información de entrada necesaria. Entonces el mensaje se envía al servidor. Este último, a su vez, siempre esperará por una petición de entrada, la procesará, y empacará los resultados en un mensaje de respuesta que enviará entonces al cliente.

Utilizar un protocolo no orientado a conexión tiene la ventaja evidente de ser eficiente. Mientras los mensajes no se pierdan o corrompan, el protocolo solicitud-respuesta esquematizado funciona bien. Desafortunadamente, implementar un protocolo tolerante a fallas ocasionales de transmisión no es algo trivial. Lo único que podemos hacer cuando el cliente no recibe un mensaje de respuesta es dejar que reenvíe la petición. Sin embargo, el problema es que el cliente no puede detectar si el mensaje de solicitud original se perdió o si la transmisión de respuesta falló. Si se perdió la respuesta, entonces reenviar una petición puede resultar en que se realice la operación dos veces. Si la operación fue algo como "transfiere $10 000 de mi cuenta bancaria", evidentemente hubiera sido mejor entonces que simplemente se reportara un error. Por otra parte, si la operación fue "dime cuánto dinero me queda", hubiera sido perfectamente aceptable reenviar la solicitud. Cuando una operación puede repetirse muchas veces sin ocasionar daño alguno, se dice que es **idempotente**. Debido a que algunas peticiones son idempotentes y otras no, debería quedar claro que no existe una solución única para tratar con los mensajes perdidos. Diferimos una explicación detallada sobre el manejo de fallas de transmisión hasta el capítulo 8.

Como una alternativa, muchos sistemas cliente-servidor utilizan un protocolo confiable orientado a conexión. Aunque esta solución no es completamente adecuada para una red de área local debido al relativo bajo rendimiento, funciona perfectamente bien en sistemas de área amplia donde las comunicaciones son inherentemente poco confiables. Por ejemplo, virtualmente todos los

protocolos de aplicaciones de internet se basan en conexiones confiables TCP-IP. En este caso, siempre que un cliente solicita un servicio, primero se establece una conexión al servidor antes de enviar la petición. El servidor generalmente utiliza la misma conexión para enviar el mensaje de respuesta, después de lo cual la conexión se interrumpe. El problema es que establecer e interrumpir una conexión es relativamente costoso, en especial cuando los mensajes de solicitud y respuesta son pequeños.

Aplicación de capas

A través del tiempo, el modelo cliente-servidor ha estado sujeto a muchos debates y controversias. Una de las principales cuestiones fue cómo establecer una diferencia clara entre un cliente y un servidor. No sorprende que con frecuencia ésta no exista. Por ejemplo, un servidor para una base de datos distribuida puede actuar continuamente como un cliente, ya que reenvía solicitudes a diferentes servidores de archivos responsables de implementar las tablas de la base de datos. En tal caso, el propio servidor de la base de datos no hace más que procesar consultas.

Sin embargo, si consideramos que muchas aplicaciones cliente-servidor están enfocadas en dar a los usuarios acceso a las bases de datos, mucha gente ha defendido una diferencia entre los siguientes tres niveles, siguiendo básicamente el estilo arquitectónico en capas que explicamos antes:

1. El nivel de interfaz de usuario.
2. El nivel de procesamiento.
3. El nivel de datos.

El nivel de interfaz de usuario contiene todo lo que se necesita para la interfaz directa con el usuario, tal como la administración visual. El nivel de procesamiento contiene por lo general las aplicaciones. El nivel de datos administra los datos reales sobre los que se está actuando.

Los clientes generalmente implementan el nivel de interfaz de usuario. Este nivel consiste en los programas que permiten a los usuarios finales interactuar con aplicaciones. Existe una importante diferencia en qué tan sofisticados son los programas de interfaz de usuario.

El programa más sencillo de interfaz de usuario no es más que una pantalla basada en caracteres. Tal interfaz se ha utilizado básicamente en ambientes mainframe. En los casos en que el mainframe controla toda interacción, incluso el teclado y el monitor, difícilmente puede hablarse de un ambiente cliente-servidor. Sin embargo, en muchos casos, la terminal del usuario realiza cierto procesamiento local en la forma de, digamos, repetición de golpes de teclado, o soporte de interfaces como en las que se edita una entrada completa antes de enviarla a la computadora principal.

En la actualidad, incluso en ambientes mainframe, vemos interfaces de usuario más avanzadas. En general, la máquina cliente ofrece al menos una visualización gráfica en la que se utilizan menús contextuales y descendentes, y muchos de sus controles de pantalla se manejan mediante un ratón en lugar del teclado. Ejemplos clásicos de tales interfaces incluyen las interfaces X-Windows, utilizadas en muchos ambientes UNIX, e interfaces anteriores desarrolladas para computadoras personales MS-DOS y Apple Macintosh.

Las interfaces de usuario modernas ofrecen bastante más funcionalidad al permitir que las aplicaciones compartan una sola ventana gráfica, y utilizarla para intercambiar datos a través de acciones de usuario. Por ejemplo, para eliminar un archivo, por lo general es posible mover el icono que representa a ese archivo hacia un icono que representa el bote de basura. De igual manera, muchos procesadores de palabras permiten al usuario mover texto de un documento hacia otra posición utilizando solamente el ratón. En el capítulo 3 retomaremos las interfaces de usuario.

Muchas aplicaciones cliente-servidor pueden construirse a partir de tres diferentes piezas: una parte que maneja la interacción con el usuario, otra que opera sobre una base de datos o sistema de archivos, y una parte intermedia que generalmente contiene la funcionalidad principal de una aplicación. Esta parte intermedia está ubicada lógicamente en el nivel de procesamiento. Por contraste con las interfaces de usuario y las bases de datos, no existen muchos aspectos comunes al nivel de procesamiento. Por tanto, daremos muchos ejemplos para poner en claro este nivel.

Como primer ejemplo, consideremos un motor de búsqueda en internet. Si ignoramos todos los *banners* animados, las imágenes, y otras ventanas de elegante apariencia, la interfaz de usuario de un motor de búsqueda es muy sencilla: un usuario escribe una cadena de palabras clave y posteriormente se le presenta una lista de títulos de páginas web. El soporte para esto es formado por una amplia base de datos de páginas web que han sido previamente recuperadas e indexadas. La parte central del motor de búsqueda es un programa que transforma la cadena de palabras clave del usuario en una o más consultas para la base de datos. Posteriormente, ordena de modo jerárquico los resultados en una lista, la cual transforma en una serie de páginas HTML. Dentro del modelo cliente-servidor, esta parte de recuperación de información generalmente se ubica al nivel de procesamiento. La figura 2-4 muestra esta organización.

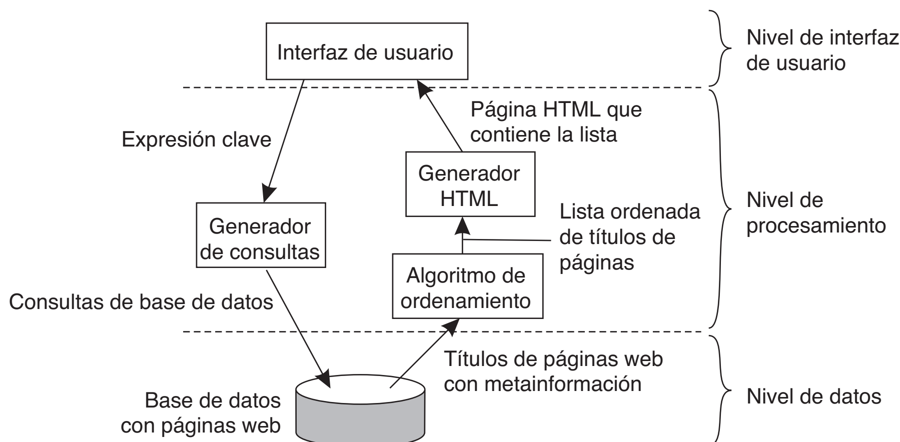


Figura 2-4. Organización simplificada, en tres capas diferentes, de un motor de búsqueda en internet.

Como segundo ejemplo, considere un sistema de soporte de decisiones implementado para un corredor de bolsa. Análogo al motor de búsqueda, un sistema como éste debe dividirse en una parte

frontal para implementar la interfaz de usuario, una parte de respaldo para acceder a la base de datos con los datos financieros, y los programas de análisis necesarios entre los dos. El análisis de datos financieros puede necesitar de sofisticados métodos y técnicas de estadística e inteligencia artificial. En algunos casos, el núcleo de un sistema de soporte de decisiones financieras incluso puede necesitar ser ejecutado en computadoras de alto rendimiento para lograr el rendimiento de procesamiento y la sensibilidad que los usuarios esperan.

Como último ejemplo, consideremos un típico paquete de escritorio consistente en un procesador de palabras, una aplicación de hoja de cálculo, interfaz de comunicación, etc. Tales paquetes de "oficina" generalmente se integran a través de una interfaz común de usuario que soporta documentos compuestos, y opera en archivos traídos desde el directorio de origen del usuario. (En un ambiente de oficina, este directorio de origen con frecuencia se ubica en un servidor de archivos remoto.) En este ejemplo, el nivel de procesamiento consiste en una colección de programas relativamente grande, cada una con capacidades simples de procesamiento.

En el modelo cliente-servidor, el nivel de datos contiene los programas que mantienen los datos reales sobre los que operan las aplicaciones. Una propiedad importante de este nivel es que los datos con frecuencia son **persistentes**, es decir, incluso si no hay aplicaciones en ejecución, los datos se almacenarán en alguna parte para su siguiente uso. En su forma más simple, el nivel de datos consiste en un sistema de archivos, pero es más común utilizar una base de datos hecha y derecha. En el modelo cliente-servidor, el nivel de datos generalmente se implementa del lado del servidor.

Además de almacenar datos, el nivel de datos generalmente es responsable de mantener datos consistentes mediante diferentes aplicaciones. Cuando se utilizan las bases de datos, mantener la consistencia significa que metadatos tales como descripciones de tablas, restricciones de acceso, y metadatos específicos de aplicaciones también se almacenan en este nivel. Por ejemplo, en el caso de un banco, podríamos querer generar una notificación cuando la deuda de la tarjeta de crédito de un cliente alcanza cierto valor. Este tipo de información puede mantenerse a través de un disparador de base de datos que active un manipulador para ese disparador en el momento adecuado.

En la mayoría de los ambientes orientados a negocios, el nivel de datos se organiza como una base de datos relacional. Aquí, la independencia de datos adquiere gran importancia. Los datos se organizan independientemente de las aplicaciones de tal manera que los cambios en la organización no afecten las aplicaciones, y que tampoco las aplicaciones afecten la organización de datos. Utilizar bases de datos relacionales en el modelo cliente servidor ayuda a separar el nivel de procesamiento del nivel de datos, cuando el procesamiento y los datos se consideran independientes.

Sin embargo, las bases de datos relacionales no siempre son la alternativa ideal. Una característica clásica de muchas aplicaciones es que operan sobre complejos tipos de datos que se modelan más fácilmente en términos de objetos que en términos de relaciones. Ejemplos de tales tipos de datos van desde simples polígonos y círculos hasta representaciones de diseños de aeronáutica, tal como en el caso de sistemas de diseño asistido por computadora (CAD, por sus siglas en inglés).

En casos donde las operaciones de datos se expresan más fácilmente en términos de manipulaciones de objetos, tiene sentido implementar el nivel de datos mediante una base de datos orientada a objetos o relacional a objetos. Es notable la forma en que este último tipo ha ganado popularidad conforme estas bases de datos se construyen sobre el modelo de datos relacional ampliamente difundido, ya que ofrecen las ventajas de la orientación a objetos.

Arquitecturas multiniveles

La diferencia entre los tres niveles lógicos que hemos visto hasta el momento sugiere cierta cantidad de posibilidades para distribuir físicamente una aplicación cliente-servidor a través de varias máquinas. La organización más simple es tener sólo dos tipos de máquina:

1. Una máquina cliente que sólo contenga los programas de implementación del nivel de interfaz de usuario.

2. Una máquina servidor que contenga el resto, es decir, los programas para implementar el nivel de procesamiento y el nivel de datos.

En esta organización el servidor maneja todo, mientras que el cliente es básicamente una Terminal tonta (simple), posiblemente con una linda interfaz gráfica. Existen muchas otras posibilidades de las cuales exploraremos unas cuantas en esta sección.

Un método efectivo para organizar clientes y servidores es distribuir los programas en capas de aplicación, explicadas en la sección anterior, a través de diferentes máquinas, como se muestra en la figura 2-5 [también consulte a Umar (1997); y Jing y cols. (1999)]. Como primer paso, diferenciamos sólo dos tipos de máquinas: máquinas cliente y máquinas servidor, lo cual nos lleva a lo que se conoce como **arquitectura de dos capas (físicamente)**.

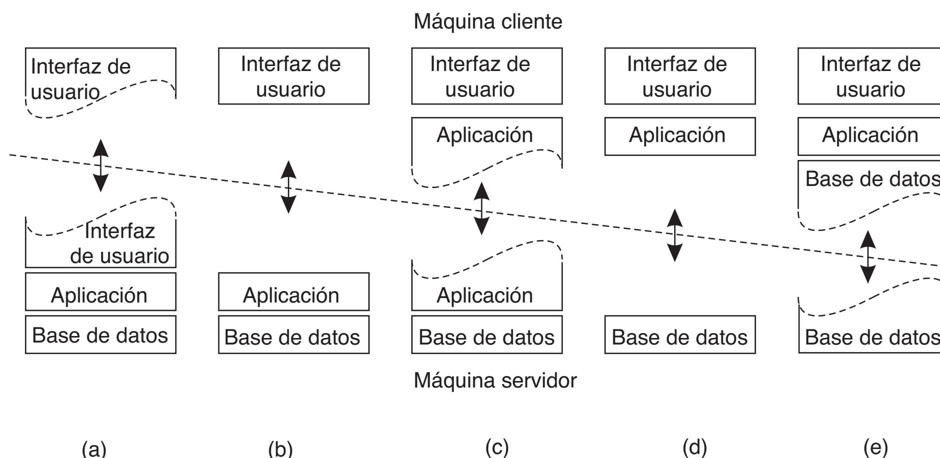


Figura 2-5. Organizaciones alternativas cliente-servidor (a) a (e).

Una organización posible es tener en la máquina cliente sólo a la parte que depende de la terminal de la interfaz de usuario, como ilustra la figura 2-5(a), y dar a las aplicaciones control remoto sobre la presentación de sus datos. Una alternativa es colocar todo el software de la interfaz de usuario del lado del cliente, según la figura 2-5(b). En tales casos, básicamente se divide la aplicación en una parte frontal gráfica, el cual se comunica con el resto de la aplicación

(residente en el servidor) a través de un protocolo específico de la aplicación. En este modelo, la parte frontal (software del cliente) no procesa más que lo necesario para presentar la interfaz de la aplicación.

Al continuar con esta línea de razonamiento, también podemos mover parte de la aplicación hacia la parte frontal, como ilustra la figura 2-5(c). Un ejemplo en el que esto tiene sentido es cuando la aplicación utiliza una forma que necesita llenarse por completo antes de procesarse. La parte frontal puede entonces verificar la corrección y consistencia de la forma, y cuando sea necesario interactuar con el usuario. Otro ejemplo de la organización de la figura 2-5(c) es un procesador de textos, donde las funciones básicas de edición se ejecutan del lado del cliente, operan localmente o sobre los datos conservados en la memoria, pero las herramientas avanzadas de soporte, como el verificador de ortografía y gramática, se ejecutan del lado del servidor.

En muchos ambientes cliente-servidor, las organizaciones que muestra la figura 2-5(d) y (e) son particularmente populares. Estas organizaciones se utilizan cuando la máquina cliente es una computadora personal o una estación de trabajo conectadas a través de una red a un sistema de archivos distribuidos o a una base de datos. En esencia, la mayoría de las aplicaciones se ejecutan en la máquina cliente, pero todas las operaciones sobre archivos o entradas de bases de datos van hacia el servidor. Por ejemplo, muchas aplicaciones bancarias se ejecutan en la máquina de un usuario final donde éste realiza transacciones y eventos similares. Una vez que el usuario termina, la aplicación contacta a la base de datos localizada en el servidor del banco y carga las transacciones para efectuar un procesamiento posterior. La figura 2-5(e) representa la situación en que el disco local del cliente contiene parte de los datos. Por ejemplo, cuando navega en la web, un cliente puede construir gradualmente en un disco local un gran depósito de las páginas web más recientemente visitadas.

Observamos que durante algunos años ha existido gran tendencia por alejarse de las configuraciones que aparecen en las figuras 2-5(d) y (e), en aquellos casos en que el software cliente se ubica en máquinas de usuarios finales. Cuando esto sucede, la mayor parte del procesamiento y almacenamiento de datos se maneja del lado del servidor. La razón para esto es simple: aunque las máquinas cliente hacen mucho, también resultan difíciles de manejar. Tener más funcionalidad en la máquina cliente hace que el software del lado del cliente sea más propenso a errores, y más dependiente de la plataforma subyacente (es decir, del sistema operativo y los recursos). Desde una perspectiva de administración de sistemas, tener lo que se conoce como **clientes obeso** no es lo óptimo. En cambio, los **clientes delgados** representados por las organizaciones que aparecen en las figuras 2-5(a) a (c) son mucho más sencillos, tal vez a un menor costo en cuanto a interfaces de usuario menos sofisticadas y a rendimiento percibido por el cliente.

Observe que esta tendencia no implica que ya no necesitemos sistemas distribuidos. Por el contrario, vemos que las soluciones del lado del servidor se están volviendo cada vez más distribuidas conforme los servidores únicos son reemplazados por varios servidores que se ejecutan en diferentes máquinas. En particular, cuando diferenciamos sólo máquinas cliente y servidor, como lo hemos hecho hasta ahora, perdemos de vista el punto de que un servidor algunas veces necesita actuar como cliente, según muestra la figura 2-6, lo cual nos lleva a una **arquitectura de tres capas (físicamente)**.

En esta arquitectura, los programas que forman parte del nivel de procesamiento residen en un servidor separado, pero también pueden estar parcialmente distribuidos en máquinas cliente y

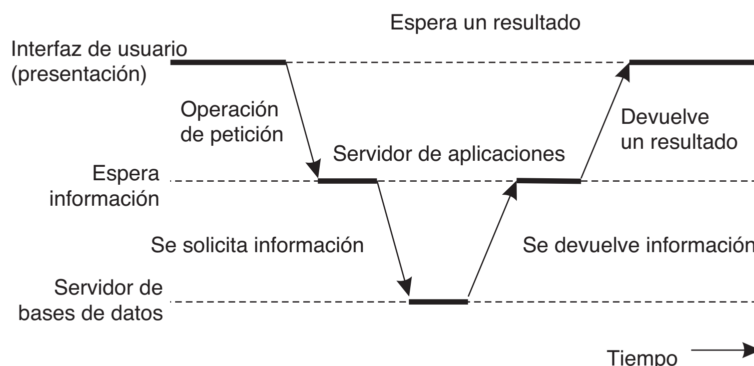


Figura 2-6. Ejemplo de un servidor que actúa como cliente.

servidor. Un ejemplo clásico de cuándo se utiliza una arquitectura de tres niveles es en el procesamiento de transacciones. Como explicamos en el capítulo 1, un proceso separado, denominado monitor de procesamiento de transacciones, coordina todas las transacciones a través de posiblemente distintos servidores de datos.

Otro ejemplo, aunque muy diferente, donde vemos una arquitectura de tres niveles, es en la organización de sitios web. En este caso, un servidor web actúa como punto de entrada a un sitio, y pasa las peticiones a un servidor de aplicaciones donde se lleva a cabo el verdadero procesamiento. Este servidor de aplicaciones, a su vez, interactúa con un servidor de bases de datos. Por ejemplo, un servidor de aplicaciones puede ser responsable de ejecutar el código necesario para inspeccionar el inventario disponible de cierta mercancía ofrecida por una librería electrónica. Para hacerlo, puede necesitar interactuar con una base de datos que contenga los datos en crudo del inventario. En el capítulo 12 retomaremos el tema de la organización de sitios Web.

2.2.2 Arquitecturas descentralizadas

Las arquitecturas cliente-servidor multiniveles son una consecuencia directa de dividir aplicaciones para obtener una interfaz de usuario, componentes de procesamiento, y un nivel de datos. Los diferentes niveles corresponden directamente a la organización lógica de las aplicaciones. En la mayor parte de los ambientes de negocios, el procesamiento distribuido equivale a organizar una aplicación cliente-servidor en la forma de una arquitectura multiniveles. Nos referimos a este tipo de distribución como **distribución vertical**. La característica clásica de la distribución vertical es que se logra colocando *lógicamente* los diferentes componentes en diferentes máquinas. El término se relaciona con el concepto de **fragmentación vertical** según es utilizado en bases de datos relacionales distribuidas, donde significa que las tablas se dividen por ancho de columna, y posteriormente se distribuyen a través de diversas máquinas (Oszu y Valduriez, 1999).

De nuevo, desde un punto de vista de administración de sistemas, tener una distribución vertical puede ayudar puesto que: las funciones están lógica y físicamente divididas en diversas

máquinas, y cada máquina se configura a la medida para un grupo específico de funciones. Sin embargo, la distribución vertical es sólo una manera de organizar aplicaciones cliente-servidor. En arquitecturas modernas, con frecuencia lo que cuenta es la distribución de clientes y servidores, y a esto nos referimos como **distribución horizontal**. En este tipo de distribución, un cliente o un servidor pueden dividirse físicamente en partes lógicas equivalentes, pero cada parte opera en su propio espacio del conjunto de datos, lo que equilibra la carga. En esta sección veremos un tipo de arquitectura moderna, llamada **sistemas punto a punto**, que da soporte a la distribución horizontal.

Desde una perspectiva de alto nivel, todos los procesos que constituyen un sistema de punto a punto son iguales. Esto significa que las funciones que necesitan realizarse están representadas por cada proceso constituyente del sistema distribuido. Como consecuencia, mucha de la interacción ocurrida entre los procesos es simétrica: cada proceso actuará como cliente y servidor al mismo tiempo (a esto se le denomina también que actúa como **sirviente**).

Dado este comportamiento simétrico, las arquitecturas de punto a punto evolucionan alrededor de la forma de cómo organizar los procesos en una **red sobrepuesta**, es decir, una red en la que los nodos estén formados por los procesos, y los vínculos representan los posibles canales de comunicación (a los cuales generalmente se les conoce como conexiones TCP). En general, un proceso no puede comunicarse directamente con otro proceso cualquiera, en vez de eso, necesita enviar mensajes a través de los canales de comunicación disponibles. Existen dos tipos de redes sobrepuestas: las que están estructuradas, y las que no lo están. Estos dos tipos son ampliamente tratados en Lua y colaboradores (2005), junto con muchos ejemplos. Aberer y colaboradores (2005) proporciona una arquitectura de referencia que permite hacer una comparación más formal de los diferentes tipos de sistemas de punto a punto. Androutsellis-Theotokis y Spinellis (2004) proporcionan una explicación desde una perspectiva de distribución de contenidos.

Arquitecturas estructuradas de punto a punto

En una arquitectura estructurada de punto a punto, la red sobrepuesta se construye mediante un procedimiento determinista. Por mucho, el procedimiento más utilizado es el de organizar los procesos a través de una **tabla hash distribuida (DHT)**. En un sistema basado en una DHT, a los elementos de datos se les asigna una llave aleatoria a partir de un identificador grande, digamos un identificador de 128 o de 160 bits. De igual manera, a los nodos del sistema también se les asigna un número aleatorio a partir del propio espacio identificador. La esencia de todo sistema basado en una DHT es, entonces, implementar un esquema eficiente y determinista que sólo mapee la llave de un elemento de datos hacia el identificador de un nodo, basándose en cierta distancia métrica (Balakrishnan, 2003). Algo más importante todavía es que, al buscar un elemento de datos, la dirección de red del nodo responsable de ese elemento de datos es devuelta. En efecto, esto se logra *enrutando* una petición de un elemento de datos hacia el nodo responsable.

Por ejemplo, en el sistema Chord (Stoica y cols., 2003), los nodos están organizados lógicamente en un anillo de tal forma que un elemento de datos con llave k se mapea hacia el nodo con el identificador más pequeño $id \geq k$. El nodo se conoce como sucesor de la llave k, y se denota como *succ(k)*, según muestra la figura 2-7. Para buscar realmente un elemento de datos, una aplicación que se ejecute en un nodo cualquiera llamaría entonces a la función LOOKUP(k), la cual

posteriormente devolvería la dirección de la red de *succ(k)*. En ese punto, la aplicación puede contactar al nodo para obtener una copia del elemento de datos.

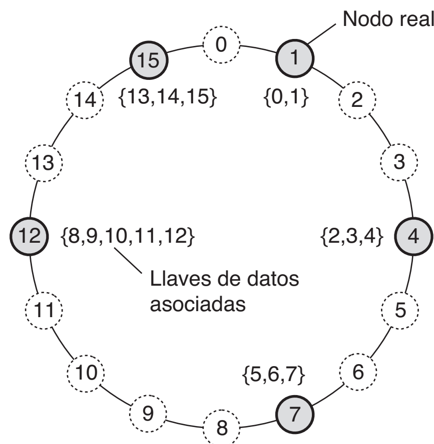


Figura 2-7. Mapeo de elementos de datos hacia nodos organizados en Chord.

Por el momento no nos adentraremos en los algoritmos necesarios para buscar una llave, pero dejaremos esta explicación para el capítulo 5 en donde describiremos los detalles de varios sistemas conocidos. Ahora concentrémonos en cómo se organizan por sí mismos los nodos en una red sobrepuesta, o, en otras palabras, en la **gestión de membresía**. En lo que veremos enseguida, es importante advertir que buscar una llave no sigue la organización lógica de nodos mostrada en el anillo de la figura 2-7. En cambio, cada nodo establecerá atajos hacia otros nodos, de tal manera que las búsquedas pueden realizarse en cierto número de pasos O(*log(N)*), donde *N* es la cantidad de nodos que participan en la superposición.

Ahora consideremos nuevamente el sistema Chord. Cuando un nodo quiere unirse al sistema, comienza por generar un identificador aleatorio, *id*. Observe que si el espacio identificador es lo suficientemente grande, entonces proporcionar el generador del número aleatorio es de buena calidad; la probabilidad de generar un identificador que ya está asignado a un nodo real es cercana a cero. Entonces, el nodo puede simplemente realizar la búsqueda en un *id*, lo cual devolverá la dirección de la red de *succ(id)*. En ese punto, el nodo de unión puede simplemente contactar a *succ(id)* y a su predecesor, e insertarse él mismo en el anillo. Desde luego, este esquema requiere que cada nodo también almacene información sobre su predecesor. La inserción provoca además que cada elemento de datos, cuya llave está ahora asociada con el nodo *id*, sea transferido desde *succ(id)*.

En términos sencillos: el *id* del nodo informa su punto de inicio a su predecesor y a su sucesor, y transfiere sus elementos de datos a *succ(id)*.

En otros sistemas basados en una DHT se siguen métodos similares. Como ejemplo, consideremos la **CAN** (*Content Addresable Network*; **red de contenido direccionable)** descrita en Ratnasamy y colaboradores (2001). CAN utiliza un espacio *d*-dimensional de coordenadas cartesianas,

el cual es dividido completamente entre todos los nodos que participan en el sistema. A manera de ejemplo, consideremos sólo el caso bidimensional que aparece en la figura 2-8.

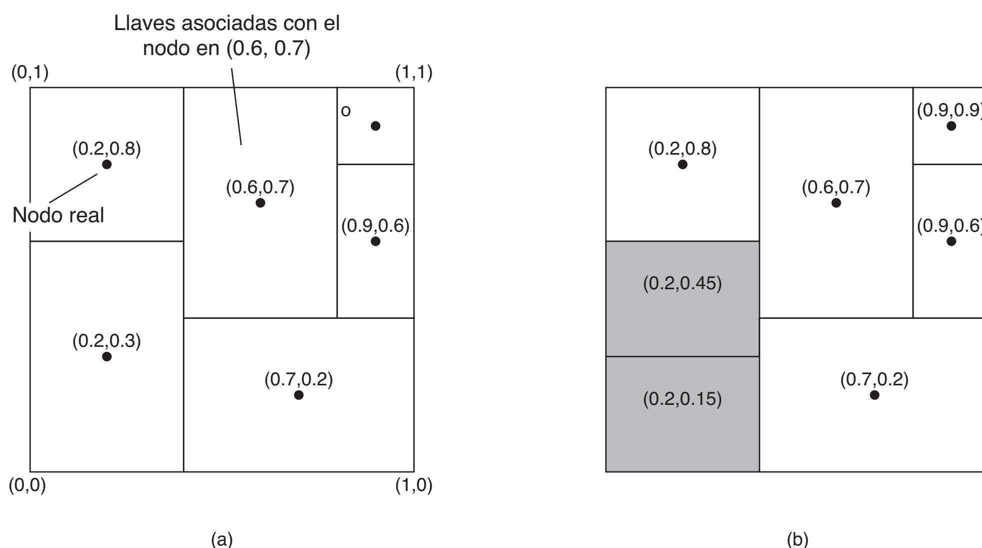


Figura 2-8. (a) Mapeo de elementos de datos hacia nodos organizados en CAN. (b) División de la región cuando un nodo se une.

La figura 2-8(a) muestra cómo el espacio bidimensional [0,1]×[0,1] se divide en seis nodos. Cada nodo tiene una región asociada. Cada elemento de datos organizado en CAN se asignará a un único punto de datos de este espacio, después de lo cual también queda claro qué nodo es responsable de ese dato (y se ignoran los elementos de datos que caen en el límite de múltiples regiones, para ello se utiliza una regla de asignación determinista).

Cuando un nodo *P* quiere unirse a un sistema CAN, éste elige un punto cualquiera del espacio coordenado, y posteriormente busca al nodo *Q* en cuya región cae ese punto. Esta búsqueda se logra a través del enrutamiento basado en posiciones, cuyos detalles dejaremos para los últimos capítulos. Después el nodo *Q* divide su región en dos mitades, como ilustra la figura 2-8(b), y una mitad se asigna al nodo *P*. Los nodos rastrean a sus vecinos, es decir, a los nodos responsables de regiones adyacentes. Cuando una región se divide, el nodo *P* que se une puede saber fácilmente quiénes son sus nuevos vecinos preguntando al nodo *Q*. Como en Chord, los elementos de datos de los que el nodo *P* ahora es responsable se transfieren desde el nodo *Q*.

Salir es un poco más problemático en CAN. Supongamos que en la figura 2-8 sale el nodo que tiene las coordenadas (0.6, 0.7). Su región será asignada a uno de sus vecinos, digamos al nodo ubicado en (0.9, 0.9), pero es claro que no puede hacerse con tan sólo fusionarla y obtener un rectángulo. En este caso, el nodo en (0.9, 0.9) simplemente se encargará de esa región e informará a los antiguos vecinos sobre este hecho. Desde luego, esto puede ocasionar una división menos simétrica del espacio coordenado, y es por esta razón que periódicamente se inicia un proceso en segundo plano para volver a dividir todo el espacio.

Arquitecturas de punto a punto no estructuradas

Los sistemas de punto a punto no estructurados se basan en algoritmos para construir redes sobrepuestas. La idea principal es que cada nodo mantenga una lista de vecinos, pero que esta lista se construya de manera más o menos aleatoria. De igual forma, se supone que los elementos de datos están colocados al azar en los nodos. Como consecuencia, cuando un nodo necesita localizar un elemento de datos específico, lo único efectivo que puede hacer es inundar la red con una consulta de búsqueda (Risson y Moors, 2006). En el capítulo 5 retomaremos la búsqueda en redes sobrepuestas no estructuradas; por ahora nos concentraremos en la gestión de membresía.

Uno de los objetivos de muchos sistemas de punto a punto no estructurados es construir una red sobrepuesta que parezca una **gráfica aleatoria**. El modelo básico es que cada nodo mantenga una lista de *c* vecinos, donde idealmente cada vecino represente un nodo *vivo* elegido al azar desde el conjunto actual de nodos. La lista de vecinos también se conoce como **vista parcial**. Existen muchas maneras de construir tales vistas parciales. Jelasity y colaboradores (2004, 2005a) desarrollaron un marco de trabajo que captura muchos algoritmos diferentes para construcción sobrepuesta para permitir evaluaciones y comparaciones. En este marco de trabajo, se supone que los nodos intercambian entradas con regularidad desde su vista parcial. Cada entrada identifica otro nodo de la red, y tiene una edad asociada que indica la edad de la referencia a ese nodo. Se utilizan dos hilos, como ilustra la figura 2-9.

El hilo activo toma la iniciativa de comunicarse con otro nodo. Éste selecciona el nodo desde su vista parcial actual. Suponiendo que las entradas necesitan ser *empujadas* hacia el par seleccionado, éste continua construyendo un buffer (búfer) que contenga $c/2+1$ entradas, incluyendo una entrada que se identifique a sí misma. Las otras entradas se toman de la vista parcial actual.

Si el nodo también se encuentra en *modo jalar*, esperará una respuesta del par seleccionado. Mientras tanto, ese par también habrá construido un buffer mediante el hilo pasivo que muestra la figura 2-9(b), y cuyas actividades se parecen bastante a las del hilo activo.

El punto crucial es la construcción de una nueva vista parcial. Esta vista, por iniciar también para el par contactado, contendrá exactamente *c* entradas, parte de las cuales provendrán del buffer recibido. En esencia, existen dos maneras de construir la nueva vista. Primero, los dos nodos pueden decidir descartar las entradas que se han enviado uno a otro. En efecto, esto significa que *intercambiarán* parte de sus vistas originales. El segundo método es descartar en lo posible a la mayor parte de entradas *viejas*. En general, se vuelve evidente que los dos métodos se complementan [para más detalles, vea Jelasity y cols. (2005a)]. Resulta que muchos protocolos de gestión de membresía para redes sobrepuestas no estructuradas van bien con este marco de trabajo. Existe cierta cantidad de observaciones por hacer.

Primero, supongamos que cuando un nodo quiere unirse contacta a cualquier otro nodo, probablemente de una lista bien conocida de puntos de acceso. Este punto de acceso es simplemente un miembro regular de la red sobrepuesta, excepto que podemos suponer que está altamente disponible. En este caso, resulta evidente que los protocolos que utilizan sólo el *modo empujar* o sólo el *modo jalar* pueden fácilmente propiciar redes sobrepuestas desconectadas. En otras palabras, los

```
Acciones de un hilo activo (repetido periódicamente):

    selecciona un par P de la vista parcial actual;
    si MODO_EMPUJAR {
        mibuffer = [(MiDireccion, 0)];
        intercambia vista parcial;
        mueve H entradas viejas hacia el final;
        agrega las primeras c/2 entradas a mibuffer;
        envia mibuffer a P;
    } sino {
        envia disparador a P;
    }
    si MODO_JALAR{
        recibe el buffer de P;
    }
construye una nueva vista parcial a partir de la actual y del buffer de P;
incrementa la edad de cada entrada en la nueva vista parcial;
```

(a)

```
Acciones de un hilo pasivo:

    recibe buffer de cualquier proceso Q;
    si MODO_JALAR{
        mibuffer = [(MiDireccion, 0)];
        intercambia vista parcial;
        mueve H entradas viejas hacia el final;
        agrega las primeras c/2 entradas a mibuffer;
        envia mibuffer a P;
    }
    construye una nueva vista parcial a partir de la actual y del buffer de P;
    incrementa la edad de cada entrada en la nueva vista parcial;
```

(b)

Figura 2-9. (a) Pasos seguidos por el hilo activo. (b) Pasos seguidos por el hilo pasivo.

grupos de nodos estarán aislados y no podrán alcanzar jamás ningún otro nodo de la red. Queda claro que ésta es una característica indeseable, razón por la cual tiene más sentido dejar que los nodos realmente *intercambien* entradas.

Segundo, abandonar la red resulta ser una operación muy simple que proporciona regularmente a los nodos vistas parciales de intercambio. En este caso, un nodo puede partir simplemente sin informarle a ningún otro nodo. Lo que ocurrirá es que cuando un nodo *P* seleccione a uno de sus supuestos vecinos, digamos el nodo *Q*, y descubra que *Q* ya no responde, simplemente eliminará la entrada de su vista parcial para seleccionar otro par. Se vuelve evidente que cuando se construye una nueva vista parcial, un nodo sigue la política de descartar tantas entradas viejas como sea

posible, y los nodos que han partido serán rápidamente olvidados. En otras palabras, las entradas que hagan referencia a nodos que han salido serán automática y rápidamente eliminadas de las vistas parciales.

Sin embargo, hay un precio a pagar cuando se sigue esta estrategia. Para explicar en qué consiste, consideremos para un nodo *P* al conjunto de nodos que, en sus vistas parciales, tienen una entrada que hace referencia a *P*. Técnicamente, a esto se le conoce como **grado de entrada** de un nodo. Entre más alto sea el grado de entrada del nodo *P*, más alta será la probabilidad de que algún otro nodo decida contactar a *P*. En otras palabras, existe cierto peligro de que *P* se vuelva un nodo popular, lo cual fácilmente provocaría una posición de desequilibrio con respecto a la carga de trabajo. Descartar sistemáticamente las entradas viejas da como resultado promover aquellos nodos que tienen un alto grado de entrada. También existen otras desventajas, para conocerlas deberemos consultar a Jelasity y colaboradores (2005a).

Administración de topología de redes sobrepuestas

Aunque parecería que los sistemas estructurados de punto a punto y no estructurados forman clases estrictamente independientes, en realidad no necesariamente es el caso [también consulte a Castro y cols. (2005)]. Una observación clave es que, al intercambiar y seleccionar cuidadosamente entradas de vistas parciales, es posible construir y mantener topologías específicas de redes sobrepuestas. Esta administración de topología se logra adoptando un método de dos capas, como ilustra la figura 2-10.

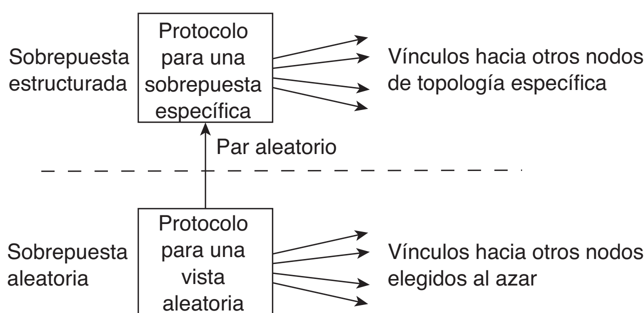


Figura 2-10. Método de dos capas para construir y mantener topologías sobrepuestas específicas utilizando técnicas de sistemas no estructurados punto a punto.

La capa más baja constituye un sistema no estructurado de punto a punto en cuyos nodos se intercambian periódicamente entradas de sus vistas parciales con la intención de mantener una gráfica aleatoria precisa. En este caso, la precisión se refiere al hecho de que la vista parcial debe llenarse con entradas que hagan referencia a nodos *vivos* seleccionados al azar.

La capa más baja pasa su vista parcial a la capa más alta, donde ocurre una selección adicional de entradas. Esto provoca entonces una segunda lista de vecinos que corresponde a la topología deseada. Jelasity y Babaoglu (2005) proponen utilizar una *función de jerarquización* mediante la cual se ordenen los nodos de acuerdo con cierto criterio relativo a un nodo dado. Una función simple de

jerarquización es para ordenar un conjunto de nodos por el incremento en la distancia desde un determinado nodo *P*. En ese caso, el nodo *P* construirá gradualmente una lista de sus vecinos más cercanos, propiciando que la capa más baja continúe pasando al azar nodos seleccionados.

Como ejemplo, considere una malla lógica de tamaño $N \times N$ con un nodo ubicado en cada punto de la malla. Se requiere que cada nodo mantenga una lista de los *c* vecinos más cercanos, en donde la distancia entre un nodo en (a_1, a_2) y (b_1, b_2) se define como $d_1 + d_2$, con $d_i = min\ (N - |a_i - b_i|, |a_i - b_i|)$. Si la capa más baja ejecuta periódicamente el protocolo que delineamos en la figura 2-9, la topología que evolucionará es un toroide, según muestra la figura 2-11.

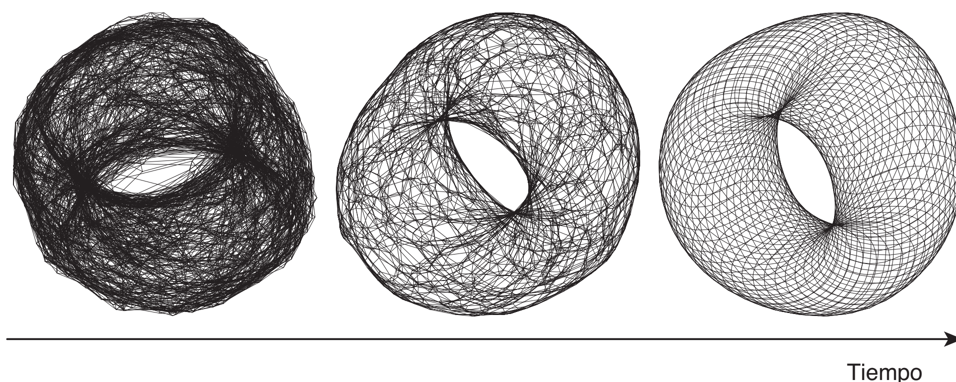


Figura 2-11. Creación de una red sobrepuesta específica mediante un sistema de igual a igual no estructurado de dos capas [adaptado con el permiso de Jelasity y Babaoglu (2005)].

Por supuesto, es posible utilizar funciones de jerarquización diferentes por completo. Son notablemente interesantes aquellas funciones relacionadas con la captura de **proximidad semántica** de los elementos de datos conforme se almacenan en un nodo par. Esta proximidad propicia la construcción de **redes sobrepuestas semánticas** que permiten utilizar algoritmos de búsqueda altamente eficientes en sistemas no estructurados de punto a punto. En el capítulo 5 retomaremos estos sistemas, cuando expliquemos la asignación de nombres basada en atributos.

Superpuntos

Es notable que en sistemas no estructurados de punto a punto, localizar elementos de datos importantes puede resultar problemático cuando la red crece. La razón de este problema de escalabilidad es sencilla: como no existe una manera determinista de enrutar una solicitud de búsqueda hacia un elemento de datos específico, la única técnica a la que un nodo puede recurrir es, esencialmente, a la acumulación de solicitudes. Existen varias formas de realizar esta acumulación, según veremos en el capítulo 5, pero como una alternativa, muchos sistemas de punto a punto han propuesto utilizar nodos especiales que mantienen un índice de elementos de datos.

Existen otras situaciones en las que abandonar la naturaleza simétrica de los sistemas de punto a punto es razonable. Consideremos una colaboración de nodos que ofrecen recursos uno a otro. Por ejemplo, en una **red de entrega de contenidos (CDN**, por sus siglas en inglés**)**, los nodos pueden ofrecer almacenamiento para hospedar copias de páginas web, lo cual permite a los clientes web acceder a páginas cercanas, y entonces acceder rápidamente a éstas. En ese caso, un nodo *P* puede necesitar buscar recursos en una parte específica de la red. Entonces, utilizar un agente que coleccione recursos útiles para un número de nodos que se encuentran en las proximidades de otros permitirá seleccionar rápidamente un nodo con recursos suficientes.

Los nodos que mantienen un índice o actúan como agentes se conocen, por lo general, como **superpuntos**. Como su nombre sugiere, los superpuntos con frecuencia también están organizados en una red de punto a punto, ello genera una organización jerárquica, según explican Yang y García-Molina (2003). Un ejemplo sencillo de tal organización aparece en la figura 2-12. En esta organización, cada punto regular se conecta como un cliente a un superpunto. Toda comunicación desde y hacia un punto regular procede a través del superpunto asociado con el punto.

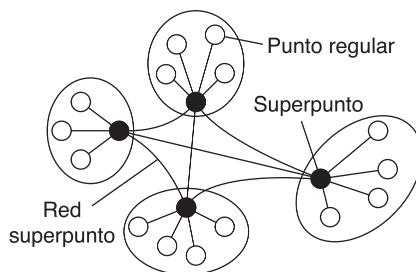


Figura 2-12. Organización jerárquica de nodos en una red de superpunto.

En muchos casos, la relación cliente-superpunto es fija: siempre que un punto regular se une a la red, se adjunta a uno de los superpunto y permanece unido a él hasta que deja la red. Evidentemente, se espera que los superpuntos sean procesos de larga vida con alta disponibilidad. Para compensar comportamientos potencialmente inestables de los superpuntos, pueden utilizarse esquemas de respaldo tales como emparejar cada superpunto con otros superpuntos y solicitar que los clientes se adjunten a ambos.

Forzar una asociación fija con un superpunto no siempre es la mejor solución. Por ejemplo, en el caso de redes de intercambio de archivos, puede resultar mejor que un cliente se adjunte a un superpunto que mantenga un índice o archivos que le interesen al cliente. En ese caso, las oportunidades son mayores que cuando el cliente busca un archivo específico, su superpunto sabrá dónde encontrarlo. Garbacki y colaboradores (2005) describe un esquema relativamente sencillo en el que la relación cliente-superpunto puede cambiar conforme los clientes descubran nuevos superpuntos con los cuales asociarse. En particular, a un superpunto que devuelve el resultado de una operación de búsqueda se le da preferencia sobre otros superpunto.

Como hemos visto, las redes de punto a punto ofrecen un medio flexible para que los nodos se unan o salgan de una red. Sin embargo, con las redes de superpunto surge un nuevo problema, a

saber, cómo seleccionar los nodos que son elegibles para volverse superpunto. Este problema está muy relacionado con el **problema de elección de líder**, el cual explicaremos en el capítulo 6 cuando retomemos la elección de superpuntos en una red de punto a punto.

2.2.3 Arquitecturas híbridas

Hasta el momento nos hemos enfocado en arquitecturas cliente-servidor y en algunas arquitecturas de punto a punto. Muchos sistemas distribuidos combinan características arquitectónicas, como hemos visto en las redes de superpuntos. En esta sección estudiaremos algunas clases específicas de sistemas distribuidos en donde las soluciones cliente-servidor se combinan con arquitecturas descentralizadas.

Sistemas de servidores al borde

Una clase importante de sistemas distribuidos organizada de acuerdo con una arquitectura híbrida, está formada por **sistemas de servidor al borde**. Estos sistemas se utilizan en internet donde los servidores se colocan "al borde" de la red. Este borde está formado por el límite que hay entre las redes empresariales y la internet real, por ejemplo, como lo proporciona un **proveedor de servicios de internet (ISP**, por sus siglas en inglés). De igual manera, donde los usuarios finales en casa se conectan a internet, a través de su ISP, podemos considerar que éste se encuentra al borde de internet. Esto nos lleva a una organización general como la que muestra la figura 2-13.

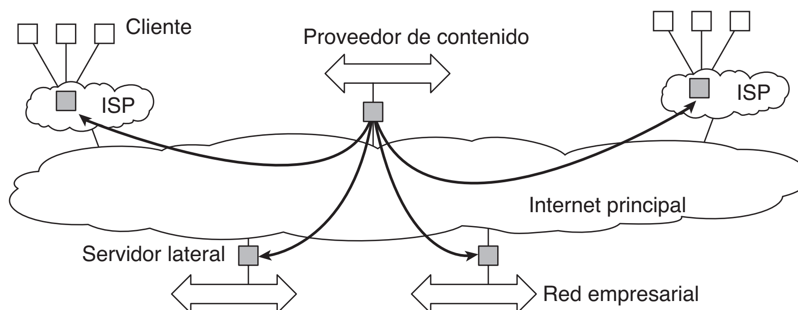


Figura 2-13. Visualización de internet como si consistiera en una colección de servidores laterales.

Los usuarios finales, o clientes en general, se conectan a internet mediante un servidor lateral. El objetivo principal del servidor lateral es proporcionar contenido, probablemente después de realizar un filtrado y descodificar funciones. Más interesante resulta el hecho de que una colección de servidores laterales pueda utilizarse para optimizar la distribución de contenido y aplicaciones. El modelo básico es que, para una organización específica, un servidor lateral actúa como servidor de origen a partir del cual se origina todo el contenido. Ese servidor puede utilizar otros servidores laterales para replicar páginas web y similares (Leff y cols., 2004; Nayate y cols., 2004; y Rabinovich y Spatscheck, 2002). En el capítulo 12 retomaremos los sistemas de servidor lateral cuando analicemos las soluciones basadas en la web.

Sistemas distribuidos en colaboración

Las estructuras híbridas se utilizan notablemente en sistemas distribuidos en colaboración. La cuestión principal en muchos de estos sistemas es que inicien por primera vez, para lo cual con frecuencia es desplegado un esquema tradicional cliente-servidor. Una vez que un nodo se ha unido al sistema, éste puede utilizar un esquema completamente descentralizado de colaboración.

Para concretar este asunto, consideremos primero el sistema de intercambio de archivos BitTorrent (Cohen, 2003). BitTorrent es un sistema de descarga de archivos de punto a punto. Su funcionamiento principal aparece en la figura 2-14. La idea básica es que cuando un usuario final busca un archivo, BitTorrent descarga partes del archivo de otros usuarios hasta que las partes descargadas pueden ensamblarse y entregar el archivo completo. Un objetivo de diseño importante era garantizar la colaboración. En la mayoría de los sistemas de intercambio de archivos, una fracción importante de participantes simplemente descarga archivos, pero prácticamente no contribuye de otro modo (Adar y Huberman, 2000; Saroiu y cols., 2003; y Yang y cols., 2005). Con este fin, un archivo puede descargarse sólo cuando el cliente que lo descarga proporciona contenido a alguien más. Pronto retomaremos este comportamiento de "una por otra".

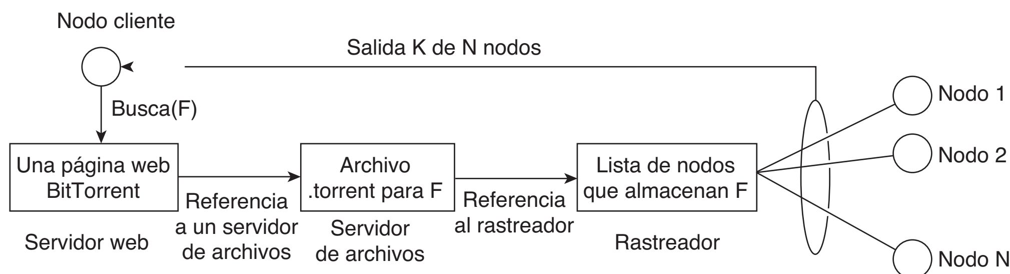


Figura 2-14. Funcionamiento principal de BitTorrent [adaptado con permiso de Pouwelse y cols. (2004)].

Para descargar un archivo, un usuario necesita el acceso a un directorio global, que es justamente uno de los muy conocidos sitios web. Tal directorio contiene referencias a algo conocido como archivos *.torrent*. Un archivo *.torrent* contiene la información necesaria para descargar un archivo específico. En particular, se refiere a lo que conocemos como **rastreador**, es decir, un servidor que mantiene una cuenta precisa de nodos *activos* que contienen secciones del archivo solicitado. Un nodo activo es uno que en el momento está descargando otro archivo. Desde luego, debe haber varios rastreadores diferentes, aunque por lo general sólo habrá un único rastreador por archivo (o colección de archivos).

Una vez que los nodos han sido identificados desde la parte donde las secciones pueden descargarse, el nodo de descarga se vuelve efectivamente activo. En ese punto, se verá forzado a ayudar a otros, por ejemplo, proporcionando secciones del archivo que está descargando, y que otros aún no tienen. Esta aplicación proviene de una regla muy sencilla: si el nodo *P* observa que el nodo *Q* descarga más de lo que sube, *P* puede decidir disminuir la velocidad a la que envía datos a *Q*.

Este esquema funciona muy bien porque *P* tiene algo qué descargar de *Q*. Por esta razón, a menudo los nodos son sustituidos con referencias a muchos otros nodos, colocándolos en una mejor posición para negociar datos.

Queda claro que BitTorrent combina soluciones centralizadas con soluciones descentralizadas. Como podemos apreciar, el cuello de botella del sistema está formado por los rastreadores.

Como otro ejemplo, considere la red de distribución de contenido Globule (Pierre y Van Steen, 2006). Globule se parece mucho a la arquitectura de servidor al borde que mencionamos antes. En este caso, en lugar de servidores al borde, usuarios finales (y también organizaciones) proporcionan voluntariamente servidores web mejorados, los cuales son capaces de colaborar en la réplica de páginas web. En su forma más simple, cada uno de estos servidores contiene los siguientes componentes:

1. Un componente que puede redirigir solicitudes del cliente hacia otros servidores.

2. Un componente para analizar patrones de acceso.

3. Un componente para administrar la réplica de páginas web.

El servidor proporcionado por Alice es el servidor web que normalmente maneja el tráfico del sitio web de Alice, y se le llama **servidor de origen** para ese sitio. Este servidor colabora con otros servidores, por ejemplo, con el proporcionado por Bob para hospedar las páginas del sitio de Bob. En este sentido, Globule es un sistema distribuido descentralizado. Solicitudes para el sitio web de Alice se reenvían inicialmente a su servidor, en donde pueden redirigirse hacia uno de los demás servidores. La redirección distribuida también está soportada.

Sin embargo, Globule también tiene un componente centralizado: su **agente**. El agente es responsable de registrar servidores y de hacer que éstos conozcan a otros. Los servidores se comunican con el agente de manera completamente análoga a lo que podría esperarse en un sistema cliente-servidor. Por razones de disponibilidad, el agente puede ser replicado, pero como veremos más adelante, este tipo de replicación se aplica ampliamente para lograr un cómputo cliente-servidor confiable.

2.3 ARQUITECTURAS *VERSUS* MIDDLEWARE

Cuando consideramos las cuestiones arquitectónicas que hemos explicado hasta este momento, una pregunta que nos viene a la mente es en dónde entra el middleware. Como explicamos en el capítulo 1, el middleware forma una capa entre las aplicaciones y las plataformas distribuidas, según muestra la figura 1-1. Un objetivo importante es proporcionar algún grado de transparencia de distribución, es decir, ocultar hasta cierto punto la distribución de los datos, el procesamiento, y el control de las aplicaciones.

Lo que vemos comúnmente en la práctica es que los sistemas middleware en realidad siguen un estilo arquitectónico específico. Por ejemplo, muchas soluciones middleware han adoptado un estilo arquitectónico basado en objetos, tal como CORBA (OMG, 2004a). Otros, como TIB/Ren-

dezvous (TIBCO, 2005) proporcionan middleware que siguen el estilo arquitectónico basado en eventos. En capítulos posteriores veremos más ejemplos de estilos arquitectónicos.

Hacer que el middleware se moldee de acuerdo con un estilo arquitectónico específico tiene el beneficio de que las aplicaciones de diseño pueden volverse más sencillas. Sin embargo, una desventaja evidente es que entonces el middleware ya no puede ser óptimo para lo que un desarrollador de aplicaciones tenía en mente. Por ejemplo, al inicio CORBA sólo ofrecía objetos que podían ser invocados por clientes remotos. Posteriormente se percibió que tener sólo esta forma de interacción resultaba demasiado restrictivo, por lo que se agregaron otros patrones de interacción tales como el intercambio de mensajes. Desde luego, agregar nuevas características fácilmente puede provocar soluciones de middleware sobrecargadas.

Además, aunque el middleware tiene como objetivo proporcionar transparencia de distribución, en general, se percibe que soluciones específicas deben ser adaptables a los requerimientos de las aplicaciones. Una solución para este problema es desarrollar diversas versiones de un sistema middleware, en donde cada versión se confeccione para una clase específica de aplicaciones. Un método que a menudo se considera mejor es hacer que los sistemas middleware sean fáciles de configurar, adaptar, y personalizar, según necesite la aplicación. Como resultado, los sistemas se desarrollan ahora de manera que la separación entre políticas y mecanismos sea más estricta. Esto ha generado diversos mecanismos mediante los cuales puede modificarse el comportamiento del middleware (Sadjadi y McKinley, 2003). Veamos algunos de los métodos más comúnmente utilizados.

2.3.1 Interceptores

De manera conceptual, un **interceptor** no es otra cosa que una construcción de software que romperá el flujo usual de control y permitirá que otro código (aplicación específica) se ejecute. Para hacer de los interceptores algo genérico se requiere un esfuerzo importante de implementación, tal como ilustran Schmidt y colaboradores (2000), y no queda claro si, en tales casos, es preferible generalizar sobre la aplicabilidad y la simplicidad. Además, en muchos casos, tener sólo capacidades limitadas de intercepción mejorará la administración del software y el sistema distribuido como un todo.

Para concretar, considere que la intercepción está soportada por muchos sistemas distribuidos basados en objetos. La idea básica es sencilla: un objeto *A* puede llamar a un método que pertenece a un objeto *B*, mientras que el objeto *B* reside en una máquina diferente de *A*. Como explicaremos más adelante, tal invocación a un objeto remoto se realiza como un método de tres pasos:

1. A un objeto *A* se le ofrece una interfaz local que es exactamente la misma que la interfaz ofrecida por el objeto *B*. *A* simplemente llama al método disponible en esa interfaz.

2. La llamada de *A* se transforma en una invocación genérica a un objeto, que se hace posible a través de una interfaz general de invocación a objetos ofrecida por el middleware de la máquina donde reside *A*.

3. Por último, la invocación genérica al objeto se transforma en un mensaje que se envía a través de la interfaz de red al nivel de transporte según ofrece el sistema operativo local de *A*.

Este esquema aparece en la figura 2-15.

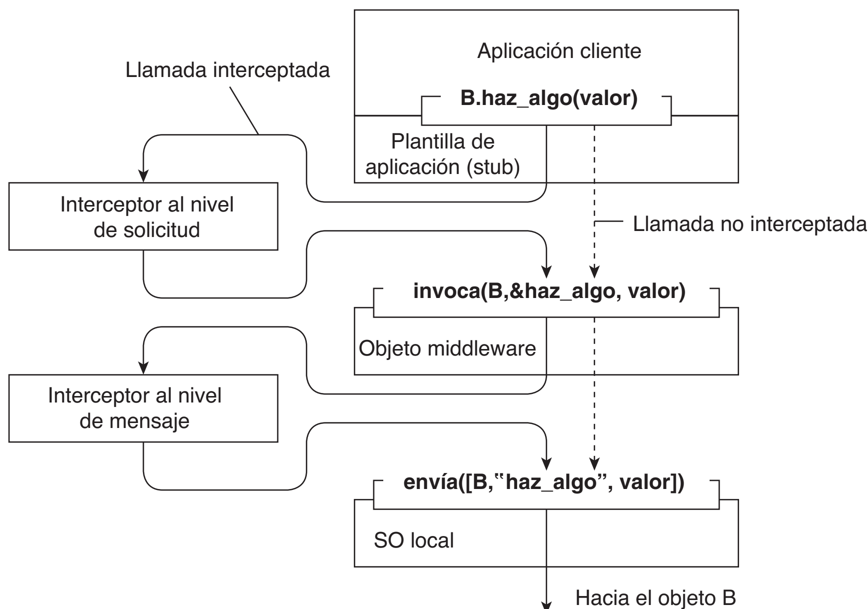


Figura 2-15. Uso de interceptores para manejar invocaciones a objetos remotos.

Después del primer paso, la llamada B.haz_algo(valor) se transforma en una llamada genérica tal como invoca(B, &haz_algo,valor) con una referencia al método de *B* y a los parámetros que van junto con la llamada. Ahora supongamos que el objeto *B* es replicado. En ese caso, cada réplica debe ser invocada realmente. Éste es un punto claro en donde la intercepción puede ayudar. Lo que el **interceptor al nivel de petición** hará es simplemente llamar a invoca(B, &haz_algo,valor) para cada una de las réplicas. La belleza de esto es que el objeto *A* no necesita darse cuenta de la réplica de *B*, y tampoco el middleware de objetos necesita componentes especiales para tratar con esta llamada replicada. Sólo el interceptor al nivel de solicitud, el cual puede *agregarse* al middleware, necesita saber sobre la replicación de *B*.

Al final, una llamada a un objeto remoto tendrá que enviarse sobre la red. En la práctica, esto significa que la interfaz de mensajes, como la ofrece el sistema operativo local, tendrá que invocarse. En ese punto, un **interceptor al nivel de mensajes** puede apoyar al transferir la invocación hacia el objeto de interés. Por ejemplo, supongamos que el parámetro valor en realidad corresponde

a un gran arreglo de datos. En ese caso, sería inteligente fragmentar los datos en partes más pequeñas para volver a juntarlas en el destino. Tal fragmentación puede mejorar el rendimiento o la confiabilidad. De nuevo, el middleware no necesita saber sobre esta fragmentación; el interceptor de nivel más bajo manejará con transparencia el resto de la comunicación con el sistema operativo local.

2.3.2 Métodos generales para software adaptativo

Lo que en realidad ofrecen los interceptores es un medio para adaptar el middleware. La necesidad de adaptación surge del hecho de que el ambiente donde se ejecutan las aplicaciones distribuidas cambia continuamente. Los cambios incluyen a aquellos resultantes de la movilidad, de una fuerte variación en la calidad del servicio de las redes, de fallas en el hardware, y de descarga de baterías, entre otros. En lugar de hacer que las aplicaciones sean responsables de reaccionar ante los cambios, este trabajo se le asigna al middleware.

Estas fuertes influencias del ambiente han llevado a los diseñadores de middleware a considerar la construcción de *software adaptativo*. Sin embargo, este software no ha tenido tanto éxito como se anticipaba. Debido a que muchos investigadores y desarrolladores lo consideran como un aspecto importante de los sistemas distribuidos modernos, analizaremos brevemente el tema. McKinley y colaboradores (2004) distinguen tres técnicas básicas para lograr la adaptación del software:

1. Separación de temas.
2. Reflexión computacional.
3. Diseño basado en componentes.

La separación de temas se relaciona con la forma tradicional de integrar a los sistemas en módulos: separar las partes que implementan la funcionalidad de aquellas encargadas de otras tareas (conocidas como *funcionalidades adicionales*) tales como la confiabilidad, el rendimiento, la seguridad, etc. Podemos afirmar que desarrollar middleware para aplicaciones distribuidas trata en gran medida con el manejo de funcionalidades adicionales independientes de las aplicaciones. El problema principal es que no se pueden separar fácilmente estas funcionalidades adicionales por medio de módulos. Por ejemplo, colocar seguridad en un módulo separado no funcionará. De igual manera, es difícil formular cómo aislar la tolerancia a fallas, ponerla en una caja, y venderla como un servicio independiente. Separar y posteriormente unir estos temas *relacionados* en un sistema (distribuido) es el tema más importante al cual se enfoca el **desarrollo de software orientado a aspectos** (Filman y cols., 2005). Sin embargo, la orientación a aspectos aún no se ha aplicado con éxito al desarrollo de sistemas distribuidos de gran escala, y al parecer todavía queda un largo camino por recorrer antes de alcanzar esta etapa.

La reflexión computacional se refiere a la habilidad de un programa para inspeccionarse a sí mismo y, si es necesario, adaptar su comportamiento (Kon y cols., 2002). La reflexión se ha construido en lenguajes de programación, incluso en Java, y ofrece una poderosa herramienta para efectuar modificaciones en tiempo de ejecución. Además, ciertos sistemas de middleware proporcionan

los medios necesarios para aplicar técnicas reflexivas. Sin embargo, así como en el caso de orientación a aspectos, el middleware reflexivo aún tiene que probarse a sí mismo como una poderosa herramienta útil para manejar la complejidad de los sistemas distribuidos de gran escala. Tal como lo mencionan Blair y colaboradores (2004), aplicar la reflexión a un amplio dominio de aplicaciones todavía está por hacerse.

Por último, el diseño basado en componentes da soporte a la adaptación a través de la composición. Un sistema puede ser configurado estáticamente al momento de diseñarlo, o dinámicamente en el tiempo de ejecución. Esto último requiere de soporte para vinculación tardía, una técnica aplicada con éxito en ambientes de lenguajes de programación y en sistemas operativos donde los módulos pueden cargarse y descargarse conforme se va necesitando. La investigación está bien encaminada ahora para permitir la selección automática de la mejor implementación de un componente durante el tiempo de ejecución (Yellin, 2003); pero, de nuevo, el proceso resulta complejo para los sistemas distribuidos, en especial cuando se considera que el reemplazo de un componente requiere saber cuál será el efecto sobre otros componentes. En muchos casos, los componentes están perdiendo independencia como es de suponer.

2.3.3 Explicación

Las arquitecturas de software para sistemas distribuidos, notablemente encontradas como middleware, son voluminosas y complejas. En gran medida, lo voluminoso y complejo surge de la necesidad de generalizar en el sentido de que es necesario proporcionar transparencia de distribución. Al mismo tiempo, las aplicaciones presentan requerimientos específicos de funcionalidad adicional, lo cual entra en conflicto con intentar lograr por completo dicha transparencia. Estos conflictivos requerimientos causados por la generalidad y la especialización han dado como resultado soluciones de middleware altamente flexibles. Sin embargo, el precio por pagar es la complejidad. Por ejemplo, Zhang y Jacobsen (2004) informan sobre un 50% de aumento en el tamaño de un producto de software en particular en sólo unos años a partir de su introducción, mientras que el total de archivos para ese producto se había triplicado durante el mismo periodo. Desde luego, ésta no es una dirección esperanzadora por promover.

Al considerar que actualmente casi todos los grandes sistemas de software requieren ejecutarse en ambientes de redes, podríamos preguntarnos si la complejidad de los sistemas distribuidos es simplemente una característica inherente al intento de lograr la transparencia de distribución. Por supuesto, cuestiones como la apertura son igualmente importantes, pero la necesidad de flexibilidad nunca ha prevalecido tanto como en el caso del middleware.

Coyler y colaboradores (2003) argumentan que lo que se necesita es un método más fuerte para implementar la simplicidad (externa), un modo más sencillo de construir middleware mediante componentes, y la independencia en las aplicaciones. Si alguna de las técnicas ya mencionadas forma parte de la solución, está sujeto a debate. En particular, ninguna de las técnicas propuestas ha encontrado hasta ahora aceptación general, ni han sido aplicadas exitosamente a sistemas de gran escala.

La suposición subyacente es que necesitamos *software adaptativo* en el sentido de que permita efectuar cambios cuando el ambiente cambie. Sin embargo, debemos preguntarnos si adaptarse a un ambiente cambiante es una buena razón para aceptar modificar el software. Hardware que falla, ataques a la seguridad, fugas de energía, etc., parecen ser influencias ambientales que pueden (y deben) ser anticipadas por el software.

El argumento más fuerte, y desde luego el más válido, para dar soporte al software adaptativo es que muchos sistemas distribuidos no pueden ser apagados. Esta restricción pide soluciones que reemplacen y mejoren componentes al momento, pero no queda claro si cualquiera de las soluciones propuestas antes es la mejor para afrontar este problema de mantenimiento.

Lo que resta entonces es que los sistemas distribuidos deben poder reaccionar a cambios en sus ambientes, por ejemplo, políticas cambiantes para destinar recursos. Todos los componentes de software implementados para permitir tal adaptación estarán en su lugar. Son los algoritmos localizados en esos componentes, y lo que dicta el comportamiento, lo que cambia sus configuraciones. El reto es dejar que tal comportamiento reactivo suceda sin intervención humana. Este método parece funcionar mejor cuando se explica la organización física de sistemas distribuidos, por ejemplo, al tomar decisiones sobre dónde colocar componentes. A continuación analizaremos tales cuestiones arquitectónicas de los sistemas.

2.4 AUTOADMINISTRACIÓN EN SISTEMAS DISTRIBUIDOS

Los sistemas distribuidos —y por supuesto su middleware asociado— deben proporcionar soluciones generales encaminadas a lograr la protección contra características indeseables inherentes a las redes, de tal manera que puedan soportar tantas aplicaciones como sea posible. Por otra parte, una total transparencia de distribución no es lo que la mayoría de las aplicaciones quiere, lo cual resulta en soluciones específicas para aplicaciones que también necesitan soporte. Hemos argumentado que, por esta razón, los sistemas distribuidos deben ser adaptativos, pero sólo cuando se trate de adaptar su comportamiento de ejecución y no los componentes de software que comprenden.

Cuando es necesario realizar la adaptación de manera automática, vemos una fuerte interrelación entre arquitecturas de sistemas y arquitecturas de software. Por una parte, necesitamos organizar los componentes de un sistema distribuido en tal forma que sea posible realizar monitoreos y ajustes, mientras que, por otra parte, requerimos decidir en dónde ejecutar los procesos de tal modo que manejen la adaptación.

En esta sección nos enfocamos explícitamente en la organización de sistemas distribuidos como sistemas de control de retroalimentación de alto nivel que permiten hacer adaptaciones automáticas a los cambios. Este fenómeno también se conoce como **cómputo autonómico** (Kephart, 2003) o **sistemas self-star** (Babaoglu y cols., 2005). El último nombre indica la variedad mediante la cual se capturan las adaptaciones automáticas: autoadministración, autoreparación, autoconfiguración, autooptimización, etc. Recurrimos a utilizar el nombre de sistemas de autoadministración para cubrir sus muchas variantes.

2.4.1 El modelo de control de retroalimentación

Existen muchas opiniones diferentes acerca de los sistemas de autoadministración, pero lo que más tienen en común (explícita o implícitamente) es la suposición de que las adaptaciones se llevan a cabo mediante uno o más **ciclos de control de retroalimentación**. Así, los sistemas organizados por medio de tales ciclos se conocen como **sistemas de control de retroalimentación**. El control de retroalimentación se ha aplicado desde hace mucho tiempo en diversos campos de la ingeniería, y sus bases matemáticas poco a poco encuentran también su camino en los sistemas de cómputo (Hellerstein y cols., 2004; y Diao y cols., 2005). Para sistemas de autoadministración, las cuestiones arquitectónicas son, de inicio, las más interesantes. La idea básica tras de esta organización es muy simple, como ilustra la figura 2-16.

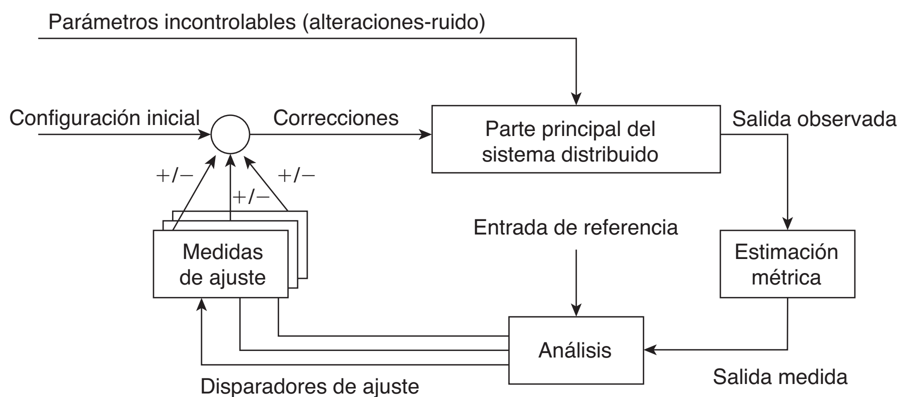


Figura 2-16. Organización lógica de un sistema de control de retroalimentación.

La parte central de un sistema de control de retroalimentación se forma con los componentes que necesitan administrarse. Se supone que estos componentes se manejan mediante parámetros de entrada controlables, pero su comportamiento puede verse influenciado por todo tipo de entrada *no controlable*, lo cual se conoce también como entrada de alteraciones o ruido. Aunque las alteraciones con frecuencia provienen del ambiente donde se ejecuta el sistema distribuido, bien puede darse el caso de que una interacción no anticipada de componentes ocasione un comportamiento inesperado.

Existen tres elementos básicos que conforman el ciclo de control de retroalimentación. Primero, el sistema mismo necesita ser monitoreado, ello requiere que varios aspectos del sistema sean medidos. En muchos casos, para medir el comportamiento es más fácil decirlo que hacerlo. Por ejemplo, en internet, los retrasos de un ciclo (envío-recepción) pueden variar mucho, y también dependen de qué tan exacta sea la medición. En tales circunstancias, estimar de modo preciso un retraso puede resultar muy difícil. Las cosas se complican aún más cuando un nodo *A* necesita estimar la latencia presente entre otros dos nodos, *B* y *C*, completamente diferentes sin interferir con ninguno. Por razones como éstas, un ciclo de control de retroalimentación generalmente contiene un **componente lógico de estimación métrica**.

Otra parte del ciclo de control de retroalimentación analiza las mediciones, y las compara con valores de referencia. Este **componente de análisis de retroalimentación** forma el núcleo del ciclo de control, ya que contendrá los algoritmos que deciden las posibles adaptaciones.

El último grupo de componentes consiste en diversos mecanismos empleados para influenciar directamente el comportamiento del sistema. Pueden existir muchos mecanismos diferentes: colocar réplicas, modificar prioridades de planificación, servicios de intercambio, mover datos por razones de disponibilidad, redirigir peticiones a diferentes servidores, etc. El componente de análisis necesita estar consciente de estos mecanismos y de sus efectos (esperados) sobre el comportamiento del sistema. Por tanto, disparará uno o varios mecanismos para, posteriormente, observar el efecto.

Una observación interesante es que el ciclo de control de retroalimentación también encaja con la administración manual de sistemas. La principal diferencia es que el componente de análisis se reemplaza con administradores humanos. Sin embargo, para administrar adecuadamente cualquier sistema distribuido, estos administradores necesitarán un equipo de monitoreo apropiado, así como mecanismos eficientes para controlar el comportamiento del sistema. Debe quedar claro que analizar adecuadamente los datos medidos y disparar las acciones correctas vuelve muy difícil el desarrollo de sistemas de autoadministración.

Debemos resaltar que la figura 2-16 muestra la organización *lógica* de un sistema de autoadministración, y como tal corresponde a lo que hemos visto cuando explicamos las arquitecturas de software. Sin embargo, la organización *física* puede ser muy diferente. Por ejemplo, el componente de análisis puede estar completamente distribuido a través del sistema. De igual manera, las mediciones de rendimiento normalmente se hacen en cada una de las máquinas que forman parte del sistema distribuido. Ahora veamos algunos ejemplos concretos sobre cómo monitorear, analizar, y corregir de manera automática sistemas distribuidos. Estos ejemplos también ilustran la diferencia que hay entre la organización lógica y la física.

2.4.2 Ejemplo: monitoreo de sistemas con Astrolabe

Como primer ejemplo, consideramos Astrolabe (Van Renesse y cols., 2003), un sistema que puede soportar el monitoreo general de sistemas distribuidos muy grandes. En el contexto de sistemas de autoadministración, Astrolabe se posicionará como una herramienta general para observar el comportamiento de sistemas. Su salida puede utilizarse para alimentar un componente de análisis que decidirá sobre acciones correctivas.

Astrolabe organiza una gran colección de servidores en cierta jerarquía por zonas. Las zonas de menor nivel constan de un solo servidor, las cuales posteriormente se agrupan en zonas de mayor tamaño. La zona de mayor nivel comprende todos los servidores. Cada servidor ejecuta un proceso Astrolabe, llamado *agente*, que colecciona información sobre las zonas en las que se encuentra ese servidor. El agente se comunica también con otros agentes con la intención de esparcir información de la zona a través de todo el sistema.

Cada servidor mantiene un conjunto de *atributos* para recopilar información local. Por ejemplo, un servidor puede rastrear archivos específicos que almacena, utilizar recursos, etc. Sólo aquellos

atributos mantenidos directamente por un servidor, es decir, al nivel más bajo de la jerarquía, pueden escribirse. Cada zona puede tener una colección de atributos, pero los valores de éstos se *calculan* a partir de los valores de las zonas de más bajo nivel.

Considere el sencillo ejemplo que muestra la figura 2-17 con tres servidores, *A*, *B* y *C*, agrupados en una zona. Cada máquina rastrea su dirección IP, la carga de la CPU, la memoria libre disponible, y el número de procesos activos. Cada uno de estos atributos puede escribirse directamente utilizando información local de cada servidor. A nivel de zona, sólo puede recopilarse información agregada, tal como el promedio de carga de la CPU o el número promedio de procesos activos.

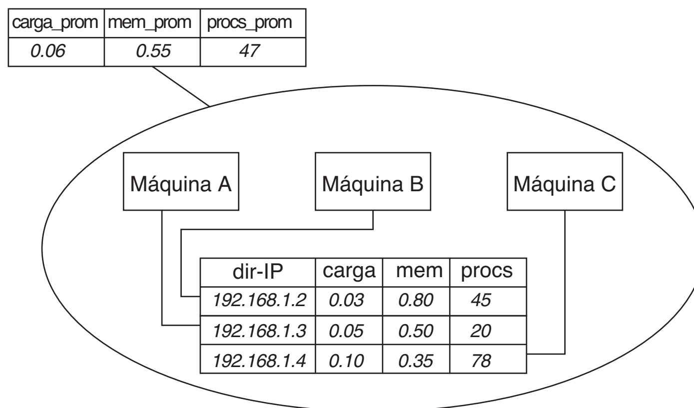


Figura 2-17. Colección de datos y agregación de información en Astrolabe.

La figura 2-17 muestra la manera en que puede verse la información reunida en cada máquina como un registro en una base de datos, y que estos registros juntos forman una relación (tabla). Esta representación se hace a propósito: es la forma en que Astrolabe ve todos los datos recopilados. Sin embargo, la información por zona sólo puede calcularse a partir de registros básicos que mantienen los servidores.

La información agregada se obtiene mediante funciones de agregación programables, las cuales son muy parecidas a las funciones disponibles del lenguaje de bases de datos relacionales SQL. Por ejemplo, suponiendo que la información del servidor de la figura 2-17 se mantiene en una tabla local llamada *hostinfo*, podríamos recopilar el número promedio de procesos para la zona que contiene las máquinas *A*, *B* y *C* mediante la sencilla consulta SQL.

```
SELECT AVG(procs) AS procs_prom FROM hostinfo
```

Combinada con algunas mejoras a SQL, no es difícil suponer que puedan formularse más consultas informativas.

Consultas como éstas son evaluadas continuamente *por* cada agente que se ejecuta en cada servidor. Desde luego, esto es posible sólo si la información de la zona se propaga a todos los nodos que

conforman Astrolabe. Con este fin, un agente que se ejecuta en un anfitrión es responsable de calcular algunas partes de las tablas de sus zonas asociadas. De manera ocasional se le envían registros, para los que no existe responsabilidad de cálculo, a través de un sencillo pero efectivo procedimiento de intercambio conocido como **gossiping**. En el capítulo 4 explicaremos con detalle los protocolos del gossiping. De igual manera, un agente pasará también resultados calculados a otros agentes.

El resultado de este intercambio de información es que, en algún momento, todos los agentes que necesiten apoyar la obtención de cierta información agregada verán el mismo resultado (porque mientras tanto no hay cambios).

2.4.3 Ejemplo: cómo diferenciar estrategias de replicación en Globule

Ahora demos un vistazo a Globule, una red colaboradora de distribución de contenido (Pierre y Van Steen, 2006). Globule se basa en servidores de usuario final colocados en internet, los cuales colaboran para optimizar el rendimiento a través de la replicación de páginas web. Para lograr esto, cada servidor de origen (es decir, el servidor utilizado para manejar las actualizaciones de un sitio web específico) rastrea patrones de acceso por página. Los patrones de acceso se expresan como operaciones de lectura y escritura para una página, y el servidor de origen para esa página registra cada operación y la marca con un tiempo.

En su forma más sencilla, Globule asume que internet puede verse como un sistema servidor puente, como ya explicamos. En particular, asume que las solicitudes siempre pueden pasar a través de un servidor puente adecuado, según muestra la figura 2-18. Este sencillo modelo permite a un servidor de origen ver qué ocurriría si colocara una réplica en un servidor puente específico. Por una parte, colocar una réplica cerca de los clientes mejoraría la espera percibida por ellos, pero induciría tráfico entre el servidor de origen y el servidor puente para mantener una réplica consistente con la página original.

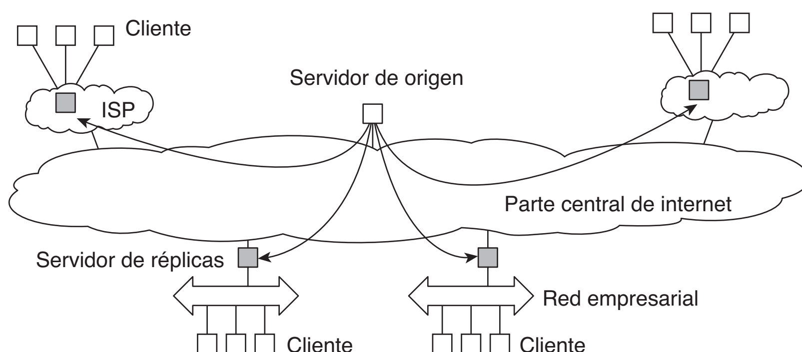


Figura 2-18. Modelo servidor puente supuesto por Globule.

Cuando un servidor de origen recibe una petición para una página, éste registra la dirección IP desde donde se originó la petición y busca al ISP o a la red empresarial asociada con la petición

utilizando el servicio de internet *WHOIS* (Deutsch y cols., 1995). El servidor de origen busca después al servidor de réplicas más cercano, que podría actuar como servidor puente para ese cliente, y posteriormente calcula la espera para ese servidor junto con el ancho de banda máximo. En su configuración más simple, Globule supone que la espera entre el servidor de réplicas y la máquina del usuario solicitante es insignificante, y de igual manera asume que el ancho de banda entre los dos es enorme.

Una vez reunidas suficientes peticiones para una página, el servidor de origen realiza un sencillo "análisis qué sucede si". Tal análisis se reduce a evaluar varias políticas de replicación, donde una política describe en dónde replicar una página y cómo mantenerla consistente. Cada política de replicación tiene un costo que puede expresarse como una función lineal simple:

$$costo = (w_1 \times m_1) + (w_2 \times m_2) + \cdots + (w_n \times m_n)$$

donde m_n denota una métrica de rendimiento y w_n es una ponderación para indicar qué tan importante es dicha métrica. Las métricas de rendimiento típicas se agregan a los retrasos entre un servidor cliente y un servidor de réplicas cuando devuelven copias de las páginas web, el ancho de banda total consumido entre el servidor de origen y el de réplicas para mantener consistente a una réplica, y el número de copias devueltas a un cliente (Pierre y cols., 2002).

Por ejemplo, suponga que el retraso típico entre el tiempo en que un cliente C hace una petición y cuando esa página es devuelta desde el mejor servidor de réplicas es d_C ms. Observe que el mejor servidor de réplicas se determina con una política de replicación. Sea m_1 el que denote el retraso agregado sobre un determinado periodo, es decir, $m_1 = \sum d_C$. Si el servidor de origen quiere optimizar la espera percibida por el cliente, elegirá un valor relativamente alto para w_1. En consecuencia, sólo las políticas que en realidad minimicen m_1 mostrarán tener costos relativamente bajos.

En Globule, un servidor de origen regularmente evalúa unas diez políticas de replicación utilizando simulación basada en rastreo para cada página web. A partir de estas simulaciones se elige la mejor política, y posteriormente se refuerza. Esto puede implicar que se instalen nuevas réplicas en diferentes servidores puente, o que se ha elegido una nueva forma de mantener consistentes a las réplicas. Coleccionar rutas, evaluar políticas de replicación, y el reforzamiento de una política seleccionada se efectúa automáticamente.

Existen unas cuantas cuestiones sutiles con las que aún tenemos que tratar. Hasta cierto punto, no queda claro cuántas peticiones tienen que recopilarse antes de que pueda realizarse la evaluación de la política en curso. Para explicarlo, suponga que en el tiempo T_i el servidor de origen selecciona la política p para el siguiente periodo hasta T_{i+1}. Esta selección ocurre de acuerdo con una serie de peticiones previas que sucedieron entre T_{i-1} y T_i. Por supuesto, en retrospectiva al tiempo T_{i+1}, el servidor puede concluir que debe seleccionar la política p^* dadas las peticiones reales que sucedieron entre T_i y T_{i+1}. Si p^* es diferente de p, entonces la selección de p en T_i estuvo mal.

Como es evidente, el porcentaje de pronósticos erróneos depende de la longitud de las series de peticiones (conocida como longitud de ruta) que se utilizan para pronosticar y seleccionar la siguiente política. Esta dependencia aparece en la figura 2-19. Lo que vemos es que el error al

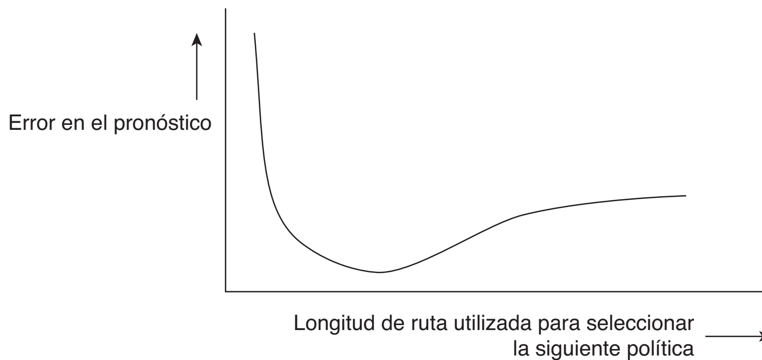


Figura 2-19. Dependencia entre la precisión en el pronóstico y la longitud de ruta.

pronosticar la mejor política aumenta si la ruta no es lo suficientemente larga. Esto se explica fácilmente por el hecho de que necesitamos las peticiones suficientes como para realizar una evaluación adecuada. Sin embargo, el error también se incrementa si utilizamos demasiadas peticiones. La razón de esto es que una longitud de ruta muy larga captura tantos *cambios* en los patrones de acceso que predecir la mejor política a seguir resulta difícil, si no es que imposible. Este fenómeno es muy conocido y es análogo a intentar pronosticar el clima de mañana analizando lo sucedido en los 100 años inmediatos previos. Es posible realizar un mejor pronóstico si sólo analizamos el pasado reciente.

Encontrar la longitud de ruta óptima también puede hacerse automáticamente; esto lo dejamos como un ejercicio para esquematizar una solución a este problema.

2.4.4 Ejemplo: administración de la reparación automática de componentes en Jade

Cuando se da mantenimiento a clusters de computadoras, cada una compuesta por sofisticados servidores, resulta importante mitigar los problemas de administración. Un método que puede aplicarse a servidores construidos bajo métodos basados en componentes es detectar las fallas latentes en ellos y reemplazarlos automáticamente. El sistema Jade sigue este enfoque (Bouchenak y cols., 2005). En esta sección lo describimos brevemente.

Jade está construido sobre el modelo de componentes Fractal, una implementación de Java de un Framework que permite agregar y eliminar componentes al tiempo de ejecución (Bruneton y cols., 2004). Un componente en Fractal puede tener dos tipos de interfaces. Una **interfaz de servidor** utilizada para llamar métodos que se implementan mediante ese componente; y una **interfaz de cliente** que utiliza un componente para llamar a otros componentes. Los componentes se conectan uno con otro mediante interfaces de **vinculación**. Por ejemplo, una interfaz de cliente del componente C_1 puede vincularse con la interfaz de servidor del componente C_2. Una primitiva de vinculación significa que una llamada a una interfaz de cliente resulta directamente en llamar a la

interfaz de servidor vinculada. En el caso de vinculación compuesta, la llamada puede proceder a través de uno o más componentes, por ejemplo, debido a que la interfaz de cliente y la interfaz de servidor no coincidieron y es necesario entonces implementar algún tipo de conversión. Otra razón puede ser que los componentes conectados se encuentren en máquinas diferentes.

Jade utiliza la idea de un **dominio de administración de reparaciones**. Tal dominio consiste en cierto número de nodos, donde cada nodo representa un servidor junto con los componentes que éste ejecuta. Aparte existe un nodo administrador responsable de agregar y eliminar nodos del dominio. El nodo administrador puede replicarse para garantizar una alta disponibilidad.

Cada nodo está equipado con detectores de fallas, los cuales monitorean el bienestar de un nodo o de uno de sus componentes e informan sobre cualquier falla al nodo administrador. En general, estos detectores consideran cambios excepcionales en el estado de los componentes, el manejo de recursos, y la falla real de un componente. Observe que lo último puede significar, en realidad, que una máquina se ha estropeado.

Cuando una falla es detectada, se inicia un procedimiento de reparación. Tal procedimiento es dirigido por una política de reparación parcialmente ejecutada por el nodo administrador. Las políticas se establecen explícitamente y se llevan a cabo de acuerdo con la falla detectada. Por ejemplo, supongamos que se detectó la falla de un nodo. En ese caso, la política de reparación puede indicar que se realicen los siguientes pasos:

1. Finalizar toda vinculación entre un componente en un nodo sin problemas y un componente en el nodo que ha fallado.
2. Solicitar al nodo administrador que inicie y agregue un nuevo nodo al dominio.
3. Configurar el nuevo nodo exactamente con los mismos componentes que se encontraban en el nodo que falló.
4. Restablecer todos los vínculos que finalizaron anteriormente.

En este ejemplo, la política de reparación es sencilla y sólo funcionará cuando no se hayan perdido datos cruciales (se dice que los componentes arruinados son **desplazados**).

El método seguido por Jade es un ejemplo de autoadministración: conforme se detecta una falla, se ejecuta automáticamente una política de reparación para llevar al sistema como un todo al estado en el que se encontraba antes del problema. Al ser un sistema basado en componentes, esta reparación automática requiere de un soporte específico para permitir que se agreguen y eliminen componentes al tiempo de ejecución. En general, no es posible cambiar aplicaciones preexistentes en sistemas de autoadministración.

2.5 RESUMEN

Los sistemas distribuidos pueden organizarse en muchas formas. Podemos diferenciar entre arquitectura de software y arquitectura de sistemas. Esta última considera en dónde se colocan, en varias

máquinas, los componentes que constituyen un sistema distribuido. La primera arquitectura tiene que ver con la organización lógica del software: cómo interactúan los componentes, en qué formas pueden estructurarse, cómo pueden hacerse independientes, etcétera.

Una idea clave cuando se habla de arquitecturas es el estilo arquitectónico. Un estilo refleja el principio básico que se sigue para organizar la interacción entre los componentes de software que conforman un sistema distribuido. Estilos importantes incluyen el modo de capas, la orientación a objetos, la orientación a eventos, y la orientación a espacios de información.

Existen muchas organizaciones diferentes para implementar los sistemas distribuidos. Una clase importante es dónde se dividen las máquinas en clientes y en servidores. Un cliente envía una petición a un servidor, el cual produce luego un resultado que es devuelto al cliente. La arquitectura cliente-servidor refleja la forma tradicional de integrar el software en módulos donde cada módulo llama las funciones disponibles en otro módulo. Al colocar diferentes componentes en diferentes máquinas, obtenemos una distribución física natural de las funciones a través de una colección de máquinas.

Las arquitecturas cliente-servidor con frecuencia están altamente centralizadas. En arquitecturas descentralizadas, a menudo vemos que los procesos que constituyen un sistema distribuido desempeñan un papel similar, lo cual también se conoce como sistemas de punto a punto. En tales sistemas, los procesos se organizan en una red sobrepuesta, la cual es una red lógica donde cada proceso tiene una lista local de otros pares con los que puede comunicarse. La red sobrepuesta puede estructurarse, en cuyo caso es posible utilizar esquemas deterministas para dirigir mensajes entre los procesos. En redes no estructuradas la lista de puntos es más o menos aleatoria, ello implica que es necesario utilizar algoritmos de búsqueda para localizar datos u otros procesos.

Como alternativa, se han desarrollado sistemas distribuidos de autoadministración. Estos sistemas combinan, hasta cierto punto, ideas de arquitecturas de sistemas y de software. Los sistemas de autoadministración, por lo general, pueden organizarse como ciclos de control de retroalimentación. Tales ciclos contienen un componente de monitoreo que mide el comportamiento del sistema distribuido, un componente de análisis para ver si es necesario ajustar algo, y una colección de diversos instrumentos para modificar el comportamiento. Los ciclos de control de retroalimentación pueden integrarse en sistemas distribuidos en muchos lugares. Aún se necesita mucha investigación para llegar a un acuerdo común sobre cómo desarrollar tales ciclos y cómo programarlos.

PROBLEMAS

1. Si un cliente y un servidor se colocan por separado, es posible advertir que la latencia de la red domina todo el rendimiento. ¿Cómo podemos afrontar este problema?

2. ¿Qué es una arquitectura cliente-servidor de tres capas?

3. ¿Cuál es la diferencia entre una distribución vertical y una horizontal?

4. Considere una cadena de procesos $P_1, P_2, \ldots, P_n$, la cual implementa una arquitectura cliente-servidor multiniveles. El proceso P_i es cliente del proceso P_{i+1}, y P_i devolverá una réplica a P_{i-1} sólo después de recibir una réplica de P_{i+1}. ¿Cuáles son los principales problemas con esta organización cuando vemos el rendimiento solicitud-respuesta en el proceso P_1?

5. En una red sobrepuesta estructurada, los mensajes se enrutan de acuerdo con la topología de la red sobrepuesta. ¿Cuál es una desventaja importante de este método?

6. Considere la red CAN de la figura 2-8. ¿Cómo enrutaría usted un mensaje desde el nodo con coordenadas (0.2, 0.3) hacia el nodo con coordenadas (0.9, 0.6)?

7. Al considerar que un nodo organizado en CAN conoce las coordenadas de sus vecinos inmediatos, una política razonable de enrutamiento sería reenviar el mensaje al nodo más cercano a su destino. ¿Qué tan buena es esta política?

8. Considere una red sobrepuesta no estructurada en la que cada nodo elige al azar c vecinos. Si P y Q son vecinos de R, ¿cuál es la probabilidad de que también sean vecinos uno de otro?

9. De nuevo, considere una red sobrepuesta no estructurada en la que cada nodo elige al azar c vecinos. Para buscar un archivo, un nodo inunda con una petición a sus vecinos y les solicita pasar la petición una vez más. ¿Cuántos nodos serán alcanzados?

10. En una red de punto a punto, no todo nodo debe volverse un superpunto. ¿Cuáles son los requerimientos razonables que debe cumplir un superpunto?

11. Considere un sistema BitTorrent en el que cada nodo tiene un vínculo de salida con una capacidad de ancho de banda B_{salida} y un vínculo de entrada con capacidad de ancho de banda $B_{entrada}$. Algunos de estos nodos (conocidos como semillas) ofrecen voluntariamente archivos para que otros nodos los descarguen. ¿Cuál es la capacidad máxima de descarga de un cliente BitTorrent si asumimos que puede contactar, cuando mucho, una semilla a la vez?

12. Proporcione un argumento convincente (técnico) del por qué la política de una por otra, como se utiliza en BitTorrent, está lejos de ser lo óptimo para compartir archivos en internet.

13. En el texto vimos dos ejemplos del uso de interceptores en middleware adaptativo. ¿Qué otros ejemplos le vienen a usted a la mente?

14. ¿Hasta qué punto dependen los interceptores del middleware en donde se utilizan?

15. Los automóviles modernos están equipados con dispositivos electrónicos. Proporcione algunos ejemplos de sistemas de control de retroalimentación instalados en ellos.

16. Proporcione un ejemplo de un sistema de autoadministración en el que el componente de análisis esté completamente distribuido o incluso oculto.

17. Esquematice una solución para determinar automáticamente la mejor longitud de ruta para pronosticar las políticas de replicación en Globule.

18. **(Asignación para el laboratorio.)** Utilice software existente para diseñar e implementar un sistema basado en BitTorrent que distribuya archivos a muchos clientes a partir de un único y poderoso servidor. Las cosas se simplifican cuando se utiliza un servidor web estándar que puede operar.

3

PROCESOS

En este capítulo, damos un vistazo de cerca a la forma en que los distintos tipos de procesos cumplen su crucial papel en los sistemas distribuidos. El concepto de proceso tiene su origen en el campo de los sistemas operativos donde, por lo general, se define como un programa en ejecución. Desde la perspectiva de un sistema operativo, la administración y la calendarización de los procesos son quizá los asuntos más importantes a tratar; sin embargo, cuando nos referimos a los sistemas distribuidos, tenemos que otros temas resultan ser igualmente o más importantes.

Por ejemplo, para organizar los sistemas cliente-servidor de manera eficiente, por lo general conviene más utilizar técnicas multihilos. Tal como explicamos en la primera sección, una de las contribuciones más importantes de los hilos a los sistemas distribuidos es que permiten la construcción de clientes y servidores de un modo en el que la comunicación y el procesamiento local se pueden traslapar, ello origina un alto nivel de rendimiento.

En los años más recientes, el concepto de virtualización ha ganado popularidad. La virtualización permite a una aplicación, e incluso a un ambiente completo incluyendo el sistema operativo, ejecutarse en forma concurrente con otras aplicaciones, pero altamente independiente de su hardware y plataforma subyacentes, provocando un alto grado de portabilidad. Más aún, la virtualización permite aislar las fallas ocasionadas por errores o problemas de seguridad. Es un concepto importante para los sistemas distribuidos, y le pondremos atención en una sección aparte.

Tal como explicamos en el capítulo 2, las organizaciones cliente-servidor son importantes en los sistemas distribuidos. En este capítulo, damos un vistazo a organizaciones comunes tanto de clientes como de servidores. También ponemos especial atención a problemas de diseño comunes para servidores.

Un tema importante, en especial para sistemas distribuidos de área amplia, es la migración de procesos entre diferentes máquinas. La migración de procesos, o más específicamente, la migración de código, puede ser útil para lograr la escalabilidad, pero también puede ayudar a configurar de manera dinámica tanto a clientes como a servidores. En este capítulo explicaremos el significado real de la migración de código y sus implicaciones.

3.1 HILOS

Aunque los procesos forman un bloque de construcción en los sistemas distribuidos, la práctica indica que la granularidad de los procesos, como tal es proporcionada por los sistemas operativos en los cuales los sistemas distribuidos son construidos, pero esto no es suficiente. En lugar de ello, resulta que tener un mayor grado de granularidad en la forma de múltiples hilos de control por proceso vuelve mucho más fácil construir aplicaciones distribuidas y obtener un mejor rendimiento. En esta sección daremos un vistazo cercano al rol que cumplen los hilos en los sistemas distribuidos y explicaremos por qué son tan importantes. En Lewis y Berg (1998) y Stevens (1999) podemos ver más sobre construcción de hilos y la manera en que se pueden utilizar.

3.1.1 Introducción a los hilos

Para entender el rol de los hilos en los sistemas distribuidos, es importante comprender lo que es un proceso y cómo se relacionan los procesos y los hilos. Para ejecutar un programa, un sistema operativo crea cierto número virtual de procesadores, cada procesador ejecuta un programa diferente. Con el fin de seguir la pista de estos procesadores virtuales, el sistema operativo tiene una **tabla de procesos** que contiene entradas para almacenar valores de los registros de la CPU, mapas de memoria, archivos abiertos, información contable, privilegios, etc. Con frecuencia, un **proceso** está definido como un programa en ejecución, esto es, un programa que es ejecutado por lo general en uno de los procesadores virtuales del sistema operativo. Una cuestión importante es que el sistema operativo tiene mucho cuidado en asegurar que los procesos independientes no puedan afectar de manera maliciosa o inadvertida la corrección del comportamiento de otro proceso. En otras palabras, garantizar que muchos procesos puedan compartir de manera concurrente la misma CPU, así como otros recursos de hardware, en forma transparente. Por lo general, el sistema operativo requiere soporte de hardware para reforzar esta separación.

Esta transparencia de concurrencia viene a un costo relativamente alto. Por ejemplo, cada vez que creamos un proceso, el sistema operativo debe crear un espacio de dirección completamente independiente. La ubicación implica inicializar segmentos de memoria mediante, por ejemplo, la puesta en cero de un segmento de datos, copiando el programa asociado dentro de un segmento de texto, y estableciendo una pila para datos temporales. De manera similar, intercambiar la CPU entre dos procesos también puede ser relativamente costoso. Aparte del ahorro en el contexto de la CPU (que consta de valores de registro, contadores de programa, apuntadores

de pila, etc.), el sistema operativo también tendrá que modificar los registros de la unidad de administración de memoria (MMU, por sus siglas en inglés) e invalidar cachés de traducción de direcciones como en el *translation lookaside buffer* (TLB). Además, si el sistema operativo da soporte a más procesos de los que puede mantener simultáneamente en memoria, pudiera requerir del intercambio de procesos entre la memoria principal y el disco antes de que el intercambio real pueda tener lugar.

Igual que un proceso, un hilo ejecuta su propio segmento de código, independientemente de otros hilos. Sin embargo, por contraste con los procesos, no se hace ningún intento por lograr un alto grado de transparencia de concurrencia si esto resulta en una degradación del rendimiento. Por tanto, un sistema de hilos mantiene generalmente sólo un mínimo de información para permitir que la CPU sea compartida por varios hilos. En especial, con frecuencia un **contexto de hilo** consta solamente del contexto de la CPU, junto con alguna otra información para el manejo de los hilos. Por ejemplo, un sistema de hilos pudiera mantener el registro de si un hilo concurrente se encuentra bloqueado dentro de una variable mútex, de modo que no sea posible seleccionarla para su ejecución. Por lo general, la información que no es estrictamente necesaria para manipular múltiples hilos es ignorada. En consecuencia, se deja por completo a los desarrolladores la protección de los datos en contra de los accesos inapropiados dentro de un proceso único.

Existen dos aplicaciones importantes de este método. Antes que nada, requerimos que el rendimiento de una aplicación multihilos rara vez sea peor que su contraparte de un solo hilo. De hecho, en muchos casos, el sistema multihilos produce ganancia en el rendimiento. En segundo lugar, debido a que los hilos no están protegidos de manera automática en contra de ellos mismos como lo están los procesos, el desarrollo de aplicaciones multihilos requiere de un trabajo intelectual adicional. Diseñar de manera apropiada y mantener las cosas sencillas resulta de gran ayuda. Desafortunadamente, la práctica común no demuestra que este principio esté bien comprendido.

Uso de hilos en sistemas no distribuidos

Antes de explicar el rol de los hilos en los sistemas distribuidos, consideremos su uso en sistemas tradicionales no distribuidos. Existen distintos beneficios de los procesos multihilos que han incrementado la popularidad hacia el uso de sistemas mediante hilos.

El principal beneficio proviene de que en un proceso con un único hilo de control, siempre que se ejecuta una llamada de sistema se bloquea el proceso como un todo. Para ejemplificar, consideremos una aplicación tal como una hoja de cálculo, y asumamos que un usuario desea modificar los valores de manera continua e interactiva. Una propiedad importante de un programa de hoja de cálculo es que mantiene las dependencias funcionales entre las diferentes llamadas, con frecuencia desde diferentes hojas de cálculo. Por tanto, cada vez que modificamos una celda, todas las celdas dependientes se actualizan de manera automática. Cuando un usuario modifica el valor de una sola celda, dicha modificación puede disparar grandes series de cálculos. Si existe solamente un hilo de control, el cálculo no puede proceder mientras el programa espera por la entrada. De manera similar, no es fácil proporcionar una entrada mientras se calculan las dependencias. La solución fácil es tener al menos dos hilos de control: uno para manipular la interacción con el usuario y otro para actualizar la hoja de cálculo. Mientras tanto, se podría utilizar un

tercer hilo para crear respaldos de la hoja de cálculo en disco cuando los otros dos hilos hacen su trabajo.

Otra ventaja de la tecnología multihilos es que hace posible explotar el paralelismo cuando ejecutamos el programa dentro de un sistema multiprocesador. En ese caso, a cada hilo se le asigna una CPU diferente mientras los datos compartidos se almacenan en una memoria principal compartida. Cuando lo diseñamos de manera apropiada, dicho paralelismo puede ser transparente: el proceso se ejecutará igual de bien en un sistema uniproceso, pero un poco más lento. La tecnología multiproceso para el paralelismo ha adquirido cada vez más importancia con la disponibilidad de estaciones de trabajo multiproceso relativamente baratas. Tales sistemas de cómputo son usados típicamente en la ejecución de servidores y de aplicaciones cliente-servidor.

La tecnología multihilos también resulta muy útil en el contexto de las grandes aplicaciones. Con frecuencia tales aplicaciones son desarrolladas como una colección de programas cooperativos para que cada programa sea ejecutado mediante un proceso aparte. Este método es típico de un ambiente UNIX. La cooperación entre programas se implementa por medio de mecanismos de comunicación interproceso (IPC, por sus siglas en inglés). Para sistemas UNIX, estos mecanismos incluyen generalmente tuberías (*pipes*), colas de mensajes, y segmentos de memoria compartida [vea también Stevens y Rago (2005)]. La mayor desventaja de todos los mecanismos IPC es que a menudo la comunicación requiere de un excesivo intercambio de contexto, como lo ilustra la figura 3-1.

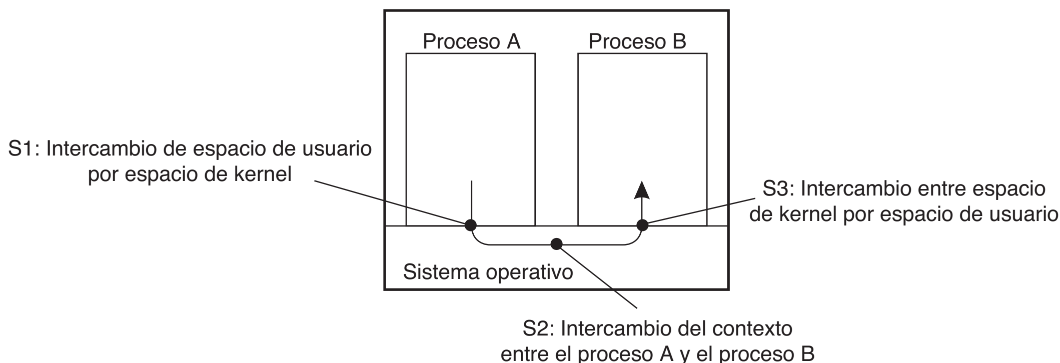


Figura 3-1. Intercambio de contexto como resultado de IPC.

Debido a que la comunicación interproceso requiere la intervención del kernel, generalmente un proceso tendrá que intercambiar primero del modo usuario al modo kernel, lo cual podemos apreciar como $S1$ en la figura 3-1. Esto requiere la modificación del mapa de memoria en la MMU, así como el reinicio del TLB. Junto con el kernel, tiene lugar un contexto de proceso ($S2$ en la figura), después de lo cual la otra parte se puede activar de nuevo mediante el intercambio entre el modo kernel al modo usuario ($S3$ en la figura 3-1). El último intercambio requiere de nuevo la modificación del mapa MMU y el reinicio del TLB.

En lugar de utilizar procesos, además puede construirse una aplicación de tal forma que las diferentes partes se ejecuten mediante hilos separados. La comunicación entre dichas partes se afronta

utilizando datos compartidos. A veces, el intercambio de hilos puede hacerse por completo en el espacio del usuario, aunque en otras implementaciones el kernel está al tanto de los hilos y los puede calendarizar. El efecto puede ser una impresionante mejora en el rendimiento.

Por último, existe una razón puramente de ingeniería de software para justificar el uso de hilos: muchas aplicaciones simplemente son más fáciles de estructurar como una colección de hilos cooperativos. Piense en aplicaciones que necesiten realizar varias tareas (más o menos independientes). Por ejemplo, en el caso de un procesador de palabras, los hilos separados se pueden utilizar para manipular la entrada del usuario, la gramática y la ortografía, la visualización del documento, la generación de índices, etcétera.

Implementación de hilos

Por lo general, los hilos son proporcionados en la forma de un paquete de hilos. Dicho paquete contiene operaciones para crear y destruir hilos, así como operaciones con respecto a la sincronización de variables tales como los mútex y las variables de condición. Existen básicamente dos métodos para implementar un paquete de hilos. El primer método es la construcción de una biblioteca de hilos que se ejecutan por completo en el modo de usuario; el segundo es que el kernel esté al tanto de los hilos y los pueda calendarizar.

Una biblioteca de hilos a nivel de usuario tiene algunas ventajas. Primero, es barato crearla y destruirla. Debido a que toda la administración de los hilos se mantiene en el espacio de direcciones del usuario, el precio de la creación de un hilo es determinado primordialmente por el costo de la ubicación de memoria para establecer una pila de hilos. De manera análoga, destruir un hilo involucra liberar la memoria para la pila, la cual ya no es utilizada. Las dos operaciones son baratas.

Otra ventaja de los hilos a nivel de usuario es que con frecuencia el intercambio de un contexto de hilo puede realizarse mediante unas cuantas instrucciones. Básicamente, sólo se requiere almacenar los valores de los registros de la CPU y, posteriormente, recargarla con valores almacenados previamente del hilo al cual se hace el intercambio. No hay necesidad de modificar los mapas de memoria, el reinicio del TLB, ni hacer un conteo de la CPU, etc. El intercambio de contextos de hilo se efectúa cuando dos hilos requieren sincronización, por ejemplo, cuando entramos a una sección de datos compartidos.

Sin embargo, una desventaja importante de los hilos a nivel de usuario es que, al invocar una llamada de bloqueo de sistema, ésta bloqueará todo el proceso al cual pertenece el hilo, y entonces bloqueará todos los hilos presentes en dicho proceso. Como ya explicamos, los hilos son particularmente útiles para estructurar grandes aplicaciones como partes que podemos ejecutar de manera lógica al mismo tiempo. En tal caso, el bloqueo de E/S pudiera no prevenir a otras partes de ser ejecutadas en la máquina. Para tales aplicaciones, los hilos a nivel usuario no ayudan.

Por lo general, estos problemas están circunscritos en su mayoría mediante la implementación de hilos dentro del kernel del sistema operativo. Desafortunadamente, se debe pagar un precio alto: cada operación de un hilo (creación, eliminación, sincronización, etc.) debe llevarse a cabo por el kernel, lo cual requiere una llamada de sistema. El intercambio de contextos de hilo puede ser ahora

tan costoso como el intercambio de contextos de proceso. Entonces, como resultado, la mayor parte de los beneficios del uso de hilos en lugar del uso de procesos desaparece.

Una solución radica en una forma híbrida entre hilos de nivel usuario y nivel kernel, por lo general se le conoce como **procesos de peso ligero** (**LWP**, por sus siglas en inglés). Un LWP se ejecuta en el contexto de un solo proceso (de peso completo), y puede haber distintos LWP por proceso. Además de contar con LWP, un sistema ofrece además un paquete de hilos a nivel usuario, lo cual permite a las aplicaciones efectuar las operaciones necesarias para crear y destruir hilos. Además, el paquete proporciona los medios para la sincronización de hilos, tales como el mútex y variables de condición. El asunto importante es que el paquete de hilos está implementado por completo en el espacio de usuario. En otras palabras, todas las operaciones sobre hilos son procesadas sin la intervención de un kernel.

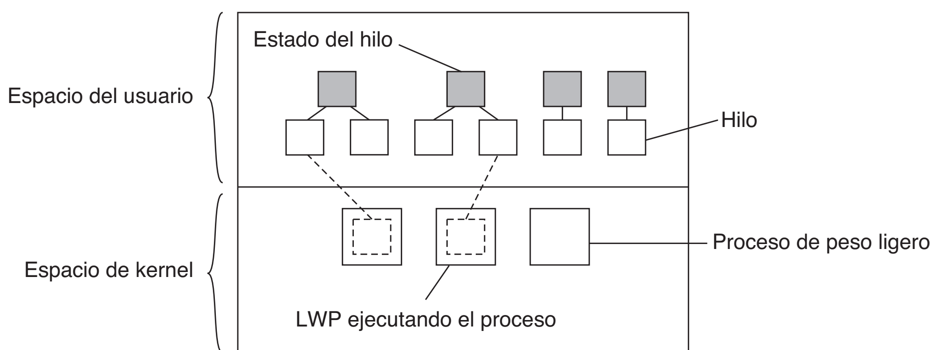


Figura 3-2. Combinación de procesos a nivel de kernel de peso ligero e hilos de nivel usuario.

Podemos compartir el paquete de hilos mediante múltiples LWP, tal como ilustra la figura 3-2. Esto significa que cada LWP puede ejecutar su propio hilo (a nivel usuario). Las aplicaciones multihilos se construyen a partir de la creación de hilos, y por consecuencia se asigna cada hilo a un LWP. Por lo general, la asignación de un hilo a un LWP está implícita y oculta del programador.

La combinación de (a nivel usuario) hilos y LWP trabaja de la siguiente manera. El paquete de hilos tiene una rutina simple para calendarizar el siguiente hilo. Al crear un LWP (lo cual se hace por medio de una llamada de sistema), el LWP cuenta con su propia pila, y tiene la instrucción de ejecutar la rutina de calendarización en busca de un hilo para ejecución. Si existen varios LWP, entonces cada uno invoca al planificador. La tabla de hilos, utilizada para seguir la pista del conjunto de hilos actuales, es compartida por los LWP. Al proteger esta tabla garantizamos que el acceso mutuamente exclusivo se lleve a cabo por medio de los mútex que se implementan por completo en el espacio de usuario. En otras palabras, la sincronización entre LWP no requiere soporte alguno del kernel.

Cuando un LWP encuentra un hilo ejecutable, cambia el contexto hacia ese hilo. Mientras tanto, otros LWP pudieran buscar otros hilos ejecutables. Si un hilo requiere bloquearse mediante un mútex o variable de condición, lleva a cabo la administración necesaria y, en algún momento, llama a la rutina de planificación. Cuando otro hilo ejecutable es localizado, se lleva a cabo un intercambio hacia dicho hilo. La belleza de todo esto es que el LWP que ejecuta el hilo no tiene que estar informado: el intercambio de contexto se implementa por completo en el espacio de usuario y ante el LWP pasa como un código de programa normal.

Ahora veamos qué sucede cuando un hilo aplica una llamada de bloqueo de sistema. En ese caso, la ejecución cambia de modo usuario a modo kernel, pero aún así continúa en el contexto del LWP actual. En el punto donde el actual LWP ya no puede continuar, el sistema operativo pudiera decidir cambiar el contexto hacia otro LWP, lo cual también implica que un cambio de contexto se regresa al modo usuario. El LWP seleccionado simplemente continuará en donde se había quedado previamente.

Existen muchas ventajas en el uso de LWP en combinación con el paquete de hilos a nivel del usuario. En primer lugar, el crear, destruir, y sincronizar hilos es relativamente más barato y no requiere intervención alguna del kernel. Segundo, en el supuesto de que un proceso no cuente con LWP suficientes, una llamada de sistema no suspenderá todo el proceso. Tercero, no hay necesidad de que una aplicación sepa algo acerca de los LWP; todo lo que ve son hilos a nivel del usuario. Cuarto, los LWP pueden utilizarse fácilmente en ambientes multiproceso, mediante la ejecución de diferentes LWP en diferentes CPU. Se puede ocultar por completo este multiproceso a la aplicación. La única desventaja de los procesos ligeros en combinación con hilos de nivel usuario es que aún requieren crear y destruir LWP, lo cual es tan caro como los hilos a nivel del kernel. Sin embargo, se requiere que la creación y destrucción de LWP sea sólo ocasional, y por lo general controlado completamente por el sistema operativo.

Una alternativa, pero similar al método del peso ligero, es hacer uso de las **activaciones del calendario** (Anderson y cols., 1991). La diferencia esencial entre las activaciones de calendario y los LWP es que cuando un hilo bloquea una llamada de sistema, el kernel hace una *llamada* hacia el paquete de hilos, llamando efectivamente a la rutina de calendarización para seleccionar el siguiente hilo ejecutable. El mismo procedimiento se repite cuando el hilo es desbloqueado. La ventaja de este método es que ahorra administración de los LWP por parte del kernel. Sin embargo, el uso de llamadas es considerado menos elegante ya que viola la estructura de los sistemas basados en capas, en los cuales solamente son permitidas las llamadas a la capa inmediata de más bajo nivel.

3.1.2 Hilos en sistemas distribuidos

Una propiedad importante de los hilos es que pueden proporcionar un medio conveniente para permitir llamadas de bloqueo de sistema sin bloquear todo el proceso en que se ejecuta el hilo. Esta propiedad vuelve a los hilos particularmente atractivos para su uso dentro de sistemas distribuidos ya que es mucho más fácil expresar la comunicación mediante múltiples conexiones lógicas al mismo tiempo. Explicaremos este punto mediante un acercamiento a los servidores y clientes multihilos, respectivamente.

Clientes multihilos

Para establecer un alto grado de transparencia de distribución, los sistemas distribuidos que operan en redes de área amplia pudieran necesitar la conciliación de grandes tiempos de propagación de mensajes de interproceso. En redes de área amplia, los ciclos tienen retrasos que pueden rondar fácilmente el orden de cientos de milisegundos, o incluso segundos en algunas ocasiones.

La manera más común de ocultar las latencias de comunicación es iniciar la comunicación y proceder de inmediato con alguna otra cosa. Un ejemplo común en donde sucede esto es en los navegadores web. En muchos casos, un documento web consta de un archivo HTML que contiene texto plano junto con una colección de imágenes, iconos, etc. Para traer cada elemento de un documento web, el navegador tiene que configurar una conexión TCP/IP, leer los datos de entrada, y pasarlos hacia el componente de visualización. Configurar la conexión, así como leer los datos de entrada que son inherentes a las operaciones de bloqueo. Al tratar con el transporte para la comunicación de grandes volúmenes, también tenemos la desventaja de que el tiempo necesario para completar cada operación podría ser relativamente largo.

Con frecuencia, un navegador web comienza con la recuperación de la página HTML y posteriormente la despliega. Para ocultar las latencias de comunicación tanto como sea posible, algunos navegadores comienzan a desplegar los datos mientras éstos siguen entrando. Mientras que el texto se pone a disposición del usuario, incluyendo las facilidades para desplazarse en sentido vertical, el navegador continúa con la recuperación de otros archivos que conforman la página, tales como las imágenes. Éstas se despliegan conforme van llegando. De esta manera, el usuario no necesita esperar hasta que todos los componentes de la página sean recuperados por completo.

En efecto, vemos que el navegador web realiza cierto número de tareas de manera simultánea. Como podemos apreciar, desarrollar los navegadores como un cliente multihilos simplifica este hecho en forma considerable. Tan pronto como el archivo HTML principal es recuperado, podemos activar hilos separados para que se hagan cargo de la recuperación de las demás partes. Cada hilo configura una conexión por separado hacia el servidor e introduce los datos. La programación de la configuración de una conexión y la lectura desde el servidor pueden llevarse a cabo mediante llamadas de sistema (bloqueo), asumiendo que la llamada de bloqueo no suspende el proceso por completo. Como explica también Stevens (1998), para cada hilo el código es el mismo y, sobre todo, simple. Mientras tanto, el usuario solamente advierte retardos durante el despliegue de las imágenes y lo demás, pero puede navegar por todo el documento.

Existe otro beneficio importante en el uso de los navegadores web multihilos en el que se pueden abrir varias conexiones de manera simultánea. En el ejemplo anterior, se configuraron varias conexiones hacia el mismo servidor. Si dicho servidor está severamente saturado, o simplemente es lento, no se observarán mejoras reales en el rendimiento comparadas con la extracción de archivos estrictamente uno después del otro.

Sin embargo, en muchos casos, los servidores web han sido replicados en distintas máquinas, donde cada servidor proporciona exactamente el mismo conjunto de documentos web. Los servidores replicados están localizados en el mismo sitio, y son conocidos con el mismo nombre. Cuando entra una petición para una página web, es reenviada a uno de los servidores, utilizando con frecuencia una estrategia *round-robin* o alguna otra técnica de balanceo de cargas (Katz y cols., 1994). Cuando utilizamos un cliente multihilos, las conexiones pueden configurarse como réplicas diferentes,

lo cual permite que los datos sean transferidos en paralelo asegurando efectivamente que se despliega la página web completa en un tiempo más corto que con un servidor no replicado. Este método es posible solamente cuando el cliente puede manipular los flujos de los datos entrantes. Los hilos son ideales para este propósito.

Servidores multihilos

Aunque existen importantes beneficios para los clientes multihilos, como hemos visto, en los sistemas distribuidos el principal uso de la tecnología multihilos está del lado del servidor. La práctica muestra que la tecnología multihilos no solamente simplifica el código del servidor de manera considerable, sino que además hace más sencillo el desarrollo de servidores que explotan el paralelismo para lograr un alto rendimiento, incluso en sistemas de un solo procesador. Sin embargo, ahora que las computadoras multiproceso están ampliamente disponibles como estaciones de trabajo de propósito general, aplicar la tecnología multihilos para implementar el paralelismo es aún más útil.

Para comprender los beneficios de los hilos para escribir código del servidor, consideremos la organización de un servidor de archivos que ocasionalmente se tiene que bloquear en espera del disco. Por lo general, el servidor de archivos espera una petición de entrada para una operación de archivo, posteriormente ejecuta la petición, y luego envía la respuesta de regreso. En la figura 3-3 podemos ver una posible y popular organización en particular. Aquí, un hilo **servidor**, lee las peticiones de entrada para una operación con archivos. Las peticiones son enviadas por clientes hacia un punto final bien conocido por este servidor. Después de examinar la petición, el hilo servidor elije un **hilo trabajador** sin utilizar (es decir, bloqueado) y le agrega la petición.

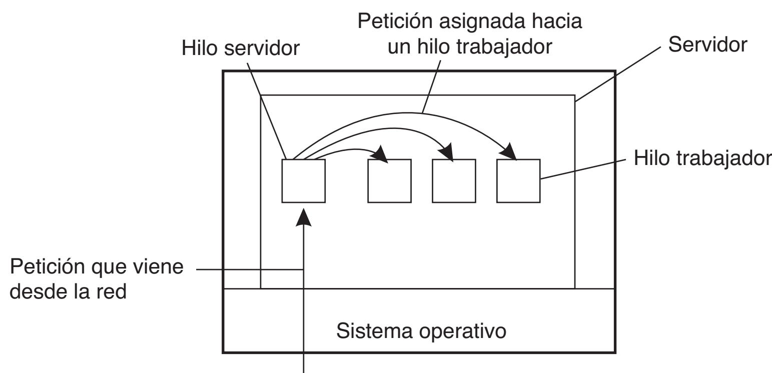


Figura 3-3. Servidor multihilos organizado en un modelo servidor/trabajador.

El trabajador procede a realizar una lectura de bloqueo en el sistema de archivos *local*, ello puede provocar que el hilo se suspenda hasta que los datos sean recuperados desde el disco. Si el hilo se suspende, se selecciona otro hilo para su ejecución. Por ejemplo, el hilo servidor puede seleccionar la adquisición de más trabajo. De manera alternativa, se puede seleccionar otro hilo trabajador que esté listo para su ejecución.

Consideremos ahora la manera en que el servidor de archivos ha sido escrito durante la ausencia de hilos. Una posibilidad es que opere como un solo hilo. El ciclo principal del servidor de archivos obtiene una petición, la examina, y la lleva a cabo hasta antes de obtener la siguiente. Mientras esperamos por el disco, el servidor está ocioso y no procesa otra petición. En consecuencia, no es posible manipular las peticiones de otros clientes. Además, si el servidor de archivos se ejecuta en una máquina dedicada, como es el caso más común, la CPU simplemente realiza una espera ociosa mientras el servidor de archivos aguarda por el disco. El resultado neto es que se pueden procesar menos peticiones por segundo. De esta manera los hilos ganan un rendimiento considerable, pero cada hilo se programa secuencialmente en forma tradicional.

Hasta el momento solamente hemos visto dos diseños posibles: un servidor de archivos multihilos y un servidor de archivos de un solo hilo. Supongamos que los hilos no están disponibles, pero los diseñadores de sistemas consideran inaceptable la pérdida de rendimiento debido al uso de un solo hilo. Una tercera posibilidad es la de ejecutar el hilo servidor como una gran máquina de estado finito. Cuando entra una petición, la examina el único hilo. Si podemos satisfacer la petición desde el caché, bien, pero si no, debemos enviar un mensaje al disco.

Sin embargo, en lugar de bloquear, el hilo servidor registra el estado de la petición actual en una tabla y luego va al siguiente mensaje. El siguiente mensaje pudiera ser o una petición para un nuevo trabajo o una respuesta desde el disco relativa a una operación previa. Si es un nuevo trabajo, éste comienza. Si es una respuesta desde el disco, se recupera tanto la información relevante desde la tabla como la respuesta procesada y posteriormente se envían al cliente. En este esquema, el servidor tendrá que hacer uso de las llamadas sin bloqueo para enviar y recibir.

En este diseño se pierde el modelo de "proceso secuencial" que teníamos en los dos primeros casos. El estado del cálculo debe ser guardado y almacenado de manera explícita en la tabla para cada mensaje enviado y recibido. En efecto, simulamos los hilos y sus pilas de la manera ruda. El proceso está operado como una máquina de estado finito que obtiene un evento y luego reacciona a él, dependiendo de lo que esté dentro.

Modelo	Características
Hilos	Paralelismo, llamadas de bloqueo de sistema
Procesos de un solo hilo	Sin paralelismo, llamadas de bloqueo de sistema
Máquina de estado finito	Paralelismo, llamadas de bloqueo de sistema

Figura 3-4. Tres maneras de construir un servidor.

Ahora debe quedar claro qué pueden ofrecer los hilos. Hacen posible retener la idea de procesos secuenciales que emiten llamadas de bloqueo de sistema (por ejemplo, un RPC para comunicarse con el disco) y aún así lograr el paralelismo. Las llamadas de bloqueo del sistema vuelven más fácil la programación, y el paralelismo mejora el rendimiento. El servidor de un solo hilo mantiene la facilidad y la simplicidad de las llamadas de bloqueo del sistema, pero cede algo del rendimiento.

El método de la máquina de estado finito logra un alto rendimiento a través del paralelismo, pero utiliza llamadas sin bloqueo, por tanto es más difícil de programar. Podemos apreciar un resumen de estos modelos en la figura 3-4.

3.2 VIRTUALIZACIÓN

Podemos ver los hilos y los procesos como una manera de hacer más cosas al mismo tiempo. En efecto, nos permiten construir (partes de) programas que parecen ejecutarse en forma simultánea. En una computadora con un solo procesador, esta ejecución simultánea es, por supuesto, una ilusión. Dado que solamente contamos con una CPU, sólo se ejecuta una instrucción de un proceso o hilo a la vez. El rápido intercambio entre hilos y procesos crea la ilusión de paralelismo.

La separación entre tener una sola CPU y ser capaz de pretender que existen más unidades de procesamiento se puede extender también a otros recursos, ello origina lo que conocemos como **virtualización de recursos**. Esta virtualización se ha aplicado durante muchas décadas, pero ha adquirido un renovado interés conforme los sistemas computacionales (distribuidos) se han vuelto más populares y complejos, originando una situación en la cual el software de aplicación por lo general sobrevive a sus sistemas de hardware y software subyacentes. En esta sección hemos puesto especial atención a lo explicado del rol de la virtualización y cómo la podemos llevar a cabo.

3.2.1 El rol de la virtualización en los sistemas distribuidos

En la práctica, cada sistema de cómputo (distribuido) ofrece una interfaz de programación hacia un nivel de software más alto, como lo muestra la figura 3-5(a). Existen muchos tipos diferentes de interfaces, abarcando desde un conjunto básico de instrucciones —como el ofrecido por una CPU— hasta una vasta colección de interfaces de programación de aplicaciones que vienen con muchos de los sistemas de middleware actuales. En su esencia, la virtualización trata con la extensión o el reemplazo de una interfaz existente de modo que imite el comportamiento de otro sistema, como vemos en la figura 3-5(b). Pronto llegaremos a la explicación de los detalles técnicos de la virtualización, pero primero nos concentraremos en el porqué es importante la virtualización para los sistemas distribuidos.

Una de las razones más importantes para introducir la virtualización en la década de 1970 fue la de permitir que el software heredado pudiera ejecutarse sobre el costoso hardware de una mainframe. El software no solamente incluía aplicaciones diversas sino que, de hecho, también los sistemas operativos estaban desarrollados para ello. Este método para soportar el software heredado se ha aplicado con éxito en las mainframes 370 de IBM (y sus sucesoras), que ofrecían una máquina virtual, las cuales soportaban diferentes sistemas operativos.

Al volverse más barato el hardware, las computadoras se hicieron más poderosas, y la cantidad de sistemas operativos de diferentes sabores se fue reduciendo, la virtualización se volvió menos que un tema. Sin embargo, por distintas razones, las cosas han cambiado desde la década de 1990, lo cual explicaremos a continuación.

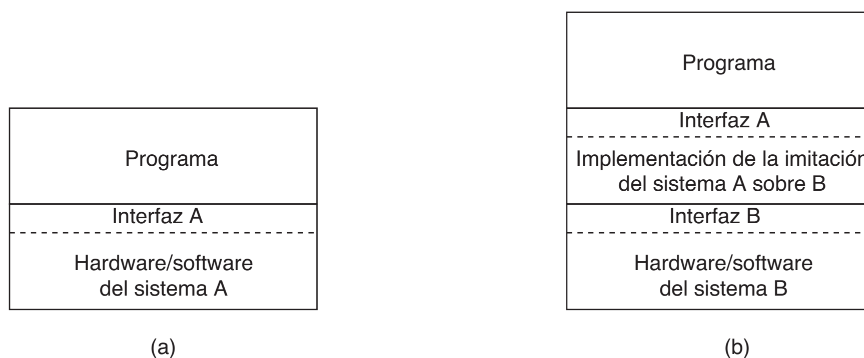


Figura 3-5. (a) Organización general entre un programa, la interfaz y el sistema. (b) Organización general entre el sistema de virtualización A sobre el sistema B.

Primero, mientras el harware y el software de los sistemas de bajo nivel cambian razonablemente rápido, el software ubicado en niveles más altos de abstracción (por ejemplo, el middleware y las aplicaciones), son mucho más estables. En otras palabras, enfrentamos una situación en la cual el software heredado no se puede mantener al mismo paso de las plataformas en que se apoya. La virtualización puede ayudar aquí al aportar las interfaces heredadas hacia nuevas plataformas, y de esta manera dar paso inmediato a lo más reciente en cuanto a grandes clases de programas existentes.

Igual de importante es el hecho de que las redes son ahora completamente masivas. Es difícil imaginar que una computadora moderna no se encuentre conectada a una red. En la práctica, esta conectividad requiere que los administradores de sistemas manejen toda una colección de servidores grandes y heterogéneos, ejecutando cada uno aplicaciones muy diferentes, a los cuales tienen acceso los clientes. Al mismo tiempo, los distintos recursos debieran ser fácilmente accesibles a estas aplicaciones. La virtualización puede ayudar mucho: la diversidad de plataformas y máquinas se puede reducir si, en esencia, dejamos que cada aplicación se ejecute en su propia máquina virtual, posiblemente incluyendo las bibliotecas relacionadas *y* el sistema operativo, el cual, a su vez, se ejecute en una plataforma común.

Este último tipo de virtualización proporciona un alto grado de portabilidad y flexibilidad; por ejemplo, con el fin de llevar a cabo en la red entregas que puedan soportar fácilmente la réplica de contenido dinámico, Awadallah y Rosenblum (2002) argumentan que la administración se vuelve más fácil cuando los servidores laterales soportan la virtualización, permitiendo tener todo un sitio, incluido su ambiente para que sea copiado de manera dinámica. Tal como explicaremos más adelante, primordialmente son esos argumentos los que hacen de la virtualización un mecanismo importante para los sistemas distribuidos.

3.2.2 Arquitecturas de máquinas virtuales

Existen diferentes formas mediante las cuales podemos ejecutar la virtualización. En el libro de Smith y Nair (2005) podemos ver una descripción de tales métodos. Para comprender las diferen-

cias en la virtualización, es importante darse cuenta de que, por lo general, los sistemas de cómputo ofrecen cuatro tipos de interfaz distintos, y en cuatro niveles diferentes:

1. Una interfaz entre el hardware y el software, constituida por **instrucciones máquina** que se pueden invocar desde cualquier programa.
2. Una interfaz entre el hardware y el software, constituida por instrucciones máquina que se pueden invocar solamente desde programas privilegiados, tales como los sistemas operativos.
3. Una interfaz que consta de **llamadas de sistema** como las que ofrece un sistema operativo.
4. Una interfaz que consta de llamadas a bibliotecas, las cuales forman, por lo general, lo que conocemos como **interfaz de programación de aplicaciones** (**API**, por sus siglas en inglés). En muchos casos, las llamadas de sistema ya mencionadas están ocultas por una API.

En la figura 3-6 mostramos los diferentes tipos de virtualización. La esencia de la virtualización es imitar el comportamiento de estas interfaces.

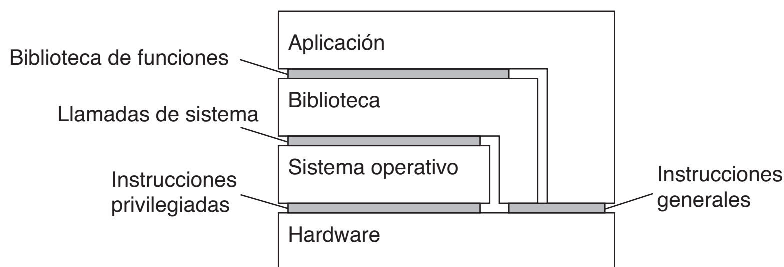


Figura 3-6. Distintas interfaces ofrecidas por los sistemas de cómputo.

La virtualización puede tener lugar en dos formas diferentes. Primero, podemos construir un sistema en tiempo de ejecución que esencialmente proporcione un conjunto de instrucciones para ser utilizado en la ejecución de aplicaciones. Las instrucciones se pueden interpretar (como en el caso del ambiente en tiempo de ejecución de Java JRE, por sus siglas en inglés), pero pudieran ser emuladas también como se hace en aplicaciones que se ejecutan en Windows sobre plataformas UNIX. Observe que en el caso referido a continuación, el emulador tendrá que imitar el comportamiento de las llamadas de sistema, las cuales han mostrado ir más allá de lo trivial. Este tipo de virtualización nos lleva a lo que Smith y Nair (2005) llaman una **máquina virtual de proceso**, lo cual enfatiza que la virtualización se implementa esencialmente solamente para un proceso.

Un método alternativo hacia la virtualización es el que proporciona un sistema que básicamente se implementa como una capa que cubre por completo al hardware original, pero que ofrece todo un conjunto de instrucciones del mismo (o de otro hardware) como una interfaz. Resulta crucial el

hecho de que se puede ofrecer esta interfaz de manera *simultánea* a diferentes programas. Como resultado, ahora es posible tener múltiples y diferentes sistemas operativos que se ejecutan de distinto modo y concurrentemente sobre la misma plataforma. A esta capa, por lo general, la conocemos como el **monitor de la máquina virtual (VMM**, por sus siglas en inglés**)**. Ejemplos típicos de este enfoque son VMware (Sugerman y cols., 2001) y Xen (Barham y cols., 2003). En la figura 3-7 se muestran estos dos métodos.

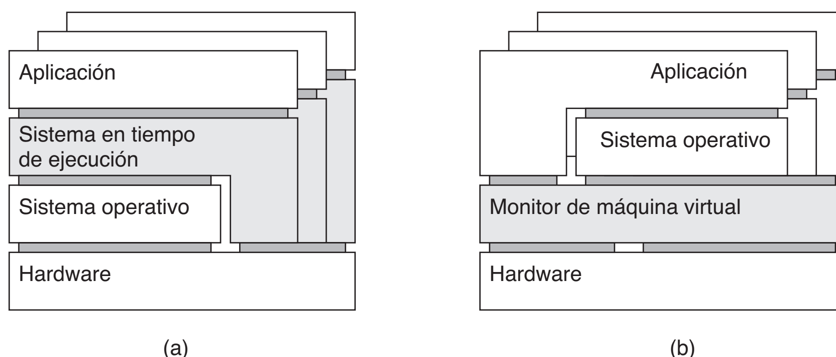


Figura 3-7. (a) Un proceso en una máquina virtual, con múltiples instancias de (aplicación, tiempo de ejecución) combinaciones. (b) Un monitoreo en una máquina virtual, con múltiples instancias de (aplicaciones, sistema operativo) combinaciones.

Tal como argumentan Rosenblum y Garfinkel (2005), las VMM serán cada vez más importantes en el contexto de la confiabilidad y la seguridad para los sistemas (distribuidos). Dado que permiten el aislamiento de toda una aplicación y de su ambiente, una falla ocasionada por un error o un ataque a la seguridad no afectará a una máquina en su totalidad. Además, como ya mencionamos, la portabilidad mejora de manera importante dado que las VMM proporcionan un desacoplamiento posterior entre hardware y software, lo cual permite mover un ambiente completo desde una máquina a otra.

3.3 CLIENTES

En los capítulos anteriores explicamos el modelo cliente-servidor, los roles de los clientes y de los servidores, y las maneras en que interactúan. Ahora veamos de cerca la anatomía de clientes y servidores, respectivamente. Comenzamos esta sección con una explicación relacionada con los clientes. En la siguiente sección explicaremos los servidores.

3.3.1 Interfaces de usuario en red

Una tarea importante de las máquinas cliente es la de proporcionar los medios necesarios para que los usuarios interactúen con servidores remotos. Existen básicamente dos maneras soportadas para

efectuar esta interacción. Primero, para cada servicio remoto, la máquina cliente tendrá una contraparte por separado que puede contactar el servicio sobre una red. Un ejemplo clásico es una agenda ejecutando en la PDA de un usuario que requiere sincronizarse con una remota, posiblemente una agenda compartida. En este caso, un protocolo a nivel de aplicación manipulará esta sincronización, como podemos ver en la figura 3-8(a).

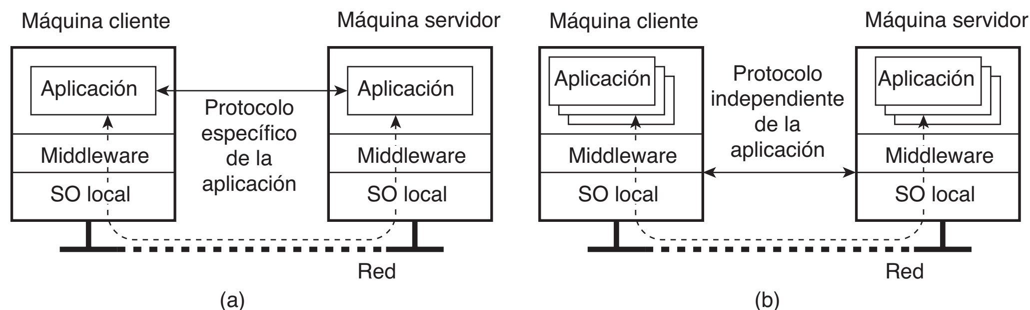


Figura 3-8. (a) Aplicación en red con su propio protocolo. (b) Solución general que permite el acceso a aplicaciones remotas.

Una segunda solución es proporcionar acceso directo a servicios remotos solamente con la oferta de una interfaz de usuario adecuada. En efecto, esto significa que la máquina cliente sólo se utiliza como terminal sin necesidad de almacenamiento local, ello produce una solución neutral para la aplicación tal como vemos en la figura 3-8(b). En el caso de interfaces de usuario en red, todo es procesado y almacenado en el servidor. Este **método de cliente delgado** recibe mayor atención al incrementarse la conectividad a internet, y los dispositivos hand-held se han vuelto cada vez más sofisticados. Como argumentamos en el capítulo anterior, las soluciones de cliente delgado también son populares ya que facilitan las tareas de administración de sistemas. Demos un vistazo a la manera en que se da soporte a las interfaces de usuario en red.

Ejemplo: Los sistemas X Window

Quizás una de las interfaces más antiguas y aún más utilizadas sea el **sistema X Window**. Por lo general, el sistema X Window, referido simplemente como X, se utiliza para controlar las terminales por mapas de bits, las cuales incluyen el monitor, el teclado, y dispositivos apuntadores tales como el ratón. En cierto sentido, podemos ver a X como parte de un sistema operativo que controla la terminal. El núcleo del sistema está formado por lo que debiéramos llamar **kernel X**. Contiene los controladores de dispositivos específicos de la terminal, y como tales, por lo general son altamente independientes del hardware.

El kernel X ofrece una interfaz de relativamente bajo nivel para controlar la pantalla, pero también para capturar los eventos del teclado y del ratón. Esta interfaz está disponible para las aplicaciones mediante una biblioteca llamada *Xlib*. Esta organización general aparece en la figura 3-9.

El aspecto interesante de X es que el kernel X y las aplicaciones X no necesariamente requieren residir en la misma máquina. En especial, X proporciona el **protocolo X**, que es un protocolo

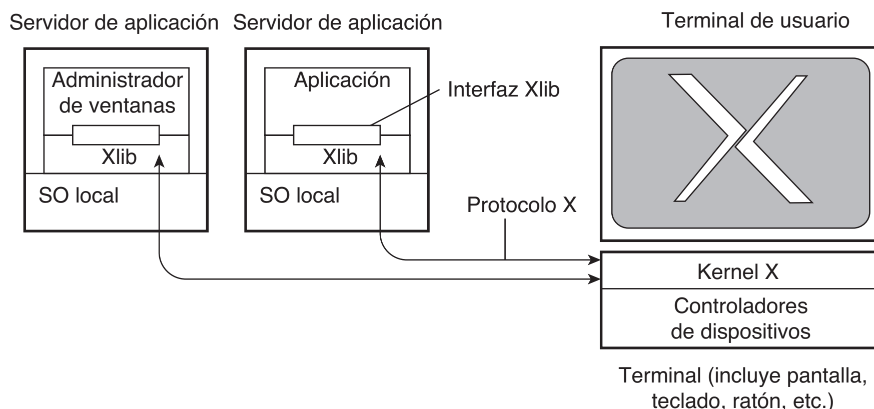


Figura 3-9. Organización básica del sistema X Window.

de comunicación a nivel de la aplicación mediante la cual una instancia de *Xlib* puede intercambiar datos y eventos con el kernel X. Por ejemplo, *Xlib* puede enviar peticiones al kernel X para crear o eliminar ventanas, establecer colores, y definir el tipo de cursor a desplegar, entre muchas otras peticiones. A cambio, el kernel X reaccionará a eventos locales tales como entradas desde el teclado y el ratón mediante el envío de paquetes de vuelta hacia *Xlib*.

Muchas aplicaciones se pueden comunicar al mismo tiempo con el kernel X. Existe una aplicación específica, a la que le están dados derechos especiales, conocida como **administrador de ventanas**. Esta aplicación puede dictar el "look and feel" de la pantalla tal como aparece ante el usuario. Por ejemplo, el administrador de ventanas puede prescribir la manera en que se decora una ventana con botones extra, la forma de colocar las ventanas en una pantalla, y así sucesivamente. Otras aplicaciones tendrán que adherirse a estas reglas.

Es interesante observar la manera en que el sistema X Window realmente encaja dentro del cómputo cliente-servidor. Hasta donde ya hemos explicado, debemos tener claro que el kernel X recibe peticiones para manipular la pantalla. Obtiene estas peticiones desde las aplicaciones (posiblemente remotas). En este sentido, el kernel X actúa como un servidor, mientras que las aplicaciones juegan el rol de clientes. Esta terminología la ha adoptado X, y aunque estrictamente hablando es correcta, puede provocar confusión.

Clientes delgados en redes de cómputo

Por evidentes razones, las aplicaciones manipulan una pantalla mediante el uso de comandos específicos de pantalla proporcionados por X. Por lo general, estos comandos se envían a través de la red en donde posteriormente el kernel X los ejecuta. Debido a su naturaleza, las aplicaciones escritas para X deben preferentemente separar la lógica de la aplicación a partir de los comandos de la interfaz de usuario. Desafortunadamente, con frecuencia éste no es el caso. Tal como han reportado Lai y Nieh (2002), resulta que mucha de la lógica de la aplicación y de la interacción con el usuario

esta íntimamente ligada, lo cual significa que una aplicación enviará muchas peticiones al kernel X, para lo cual esperará una respuesta antes de ser capaz de realizar el siguiente paso. Este comportamiento sincronizado podría afectar de manera adversa el rendimiento durante la operación sobre una red de área amplia con latencias grandes.

Existen diversas soluciones para este problema. Una es la reingeniería de la implementación del protocolo X, tal como se hizo con NX (Pinzari, 2003). Una parte importante de este trabajo se concentra en la reducción del ancho de banda mediante la compresión de mensajes X. Primero, consideramos que los mensajes constan de una parte fija, la cual se trata como un identificador, y una parte variable. En muchos casos, múltiples lenguajes tendrán el mismo identificador, en cuyo caso contendrán con frecuencia datos similares. Se puede utilizar esta propiedad para enviar solamente las diferencias entre mensajes que tengan el mismo identificador.

Tanto del lado del envío como del de la recepción se mantiene un caché local para buscar las entradas mediante el uso del identificador del mensaje. Cuando se envía un mensaje, busca primero en el caché local. Si lo encuentra, esto significa que se envió un mensaje previo con el mismo identificador pero posiblemente con diferentes datos. En ese caso, se utiliza una codificación diferencial para enviar solamente las diferencias entre los dos. Del lado de la recepción, el mensaje busca también en el caché local, después de lo cual puede emprender la decodificación de las diferencias. En el uso del caché se utilizan técnicas de compresión estándar, las cuales por lo general provocan un aumento del cuádruple del rendimiento en el ancho de banda. Sobre todo, esta técnica a reportado reducción del ancho de banda hasta en un factor de 1000, lo cual permite a X ejecutar también enlaces de bajo ancho de banda de solamente 9 600 kbps.

Un efecto colateral importante de los mensajes con caché es que del lado del envío y del lado receptor existe información compartida sobre el estado actual de la pantalla. Por ejemplo, la aplicación pudiera solicitar información geométrica relativa a distintos objetos mediante la simple petición en el caché local. Tan sólo el poseer esta información compartida reduce el número de mensajes necesarios para mantener sincronizada la aplicación y la pantalla.

No obstante estas mejoras, X aún requiere tener en ejecución un servidor de pantalla. Esto podría significar pedir demasiado, especialmente si la pantalla es algo tan simple como un teléfono celular. Una solución para mantener el software en pantalla lo más simple posible es permitir que todo el procesamiento tenga lugar del lado de la aplicación. Efectivamente, esto significa que toda la pantalla es controlada a nivel de pixeles aplicación del lado de la aplicación. Los cambios en el mapa de bits son enviados entonces desde la red hacia la pantalla, de donde se transfieren inmediatamente hacia el búfer local de la trama (frame).

Este método requiere de técnicas sofisticadas de compresión con objeto de prevenir que la disponibilidad del ancho de banda se convierta en problema. Por ejemplo, consideremos el desplegado de un flujo de video a una velocidad de 30 tramas por segundo sobre una pantalla de 320×240. Tal tamaño de pantalla es común en muchas PDA. Si cada píxel está codificado a 24 bits, sin compresión necesitaríamos entonces un ancho de banda de aproximadamente 53 Mbps. En tal caso, claramente la compresión es necesaria, y muchas técnicas de este tipo se están depurando en la actualidad. Sin embargo, observe que la compresión requiere descompresión del lado de la recepción, la cual, en cambio, puede salir cara a nivel de cómputo sin el soporte del hardware. Es posible dar soporte al hardware, pero esto eleva el costo de los dispositivos.

La desventaja de enviar datos crudos de píxeles en comparación con protocolos de alto nivel tales como X es que resulta imposible hacer uso de la semántica de la aplicación, ya que efectivamente se pierde en dicho nivel. Baratto y colaboradores (2005) proponen una técnica diferente. En su solución, referida como THINC, proporcionan unos cuantos comandos de despliegue que operan a nivel de los controladores de dispositivos de video. De esta manera estos comandos son independientes del dispositivo, más poderosos que las operaciones básicas con píxeles pero menos poderosos comparados con lo que ofrece un protocolo como X. El resultado es que los servidores de pantalla pueden ser mucho más simples, lo cual es bueno para el uso de la CPU, mientras que al mismo tiempo se pueden utilizar las optimizaciones que dependen de la aplicación para reducir en ancho de banda y la sincronización.

En THINC, las peticiones de despliegue desde la aplicación son interceptadas y traducidas dentro de comandos de más bajo nivel. Mediante la intercepción de peticiones de la aplicación, THINC puede hacer uso de la semántica de la aplicación para decidir cuál combinación de bajo nivel debemos utilizar. Los comandos traducidos no se envían de inmediato a la pantalla, en lugar de eso se forman en una fila. Al formar en un proceso por lotes distintos comandos es posible agregar comandos de pantalla dentro de una fila, ello disminuye el número de mensajes. Por ejemplo, cuando un nuevo comando para dibujar una región particular de la pantalla sobreescribe de manera efectiva lo que pudo haber establecido un comando previo (y aún en formación), este último debe ser enviado a la pantalla. Finalmente, en lugar de dejar que la pantalla solicite refrescarse, THINC siempre empuja las actualizaciones al volverse disponible. Este método de empujar ahorra latencia ya que no requiere el envío de una petición de actualización por parte de la pantalla.

Como podemos darnos cuenta, el método que sigue THINC proporciona un mejor rendimiento general, aunque mucho es acorde con el método mostrado por NX. Los detalles de la comparación del rendimiento los podemos encontrar en Baratto y colaboradores (2005).

Documentos compuestos

Las interfaces de usuario modernas hacen algo más que sistemas tales como X o simplemente sus aplicaciones. En especial, muchas interfaces de usuario permiten a las aplicaciones compartir una sola ventana gráfica, y utilizar dicha ventana para intercambiar datos a través de acciones de usuario. Las acciones adicionales que el usuario puede incluir comprenden lo que llamamos, respectivamente, operaciones de **arrastrar y soltar** y **edición y sustitución**.

Un ejemplo clásico de la funcionalidad de arrastrar y soltar es el movimiento de un icono que representa al archivo *A* hacia una representación en el icono de un cesto de basura, lo cual resulta en la eliminación del archivo. En este caso, la interfaz de usuario necesitará hacer más que solamente arreglar los iconos en la pantalla: deberá pasar el nombre del archivo *A* a la aplicación asociada con el cesto de basura tan pronto como el icono de *A* se mueva sobre la aplicación del cesto de basura. Podemos encontrar fácilmente otros ejemplos.

Es posible explicar mejor la edición y sustitución por medio de un documento que contiene texto y gráficos. Imagine que el documento es desplegado dentro de un procesador de palabras popular. Tan pronto como el usuario coloca el ratón sobre una imagen, la interfaz de usuario pasa dicha información al programa de dibujo para permitir al usuario modificar la imagen. Por ejemplo, el usuario pudo haber rotado la imagen, lo cual podría afectar la posición de la imagen en el documento.

Por tanto, la interfaz de usuario descubre la nueva altura y longitud de la imagen, y pasa esta información al procesador de palabras. Este último, a su vez, puede actualizar entonces de manera automática la vista de la página del documento.

La idea clave tras de estas interfaces de usuario es la de un **documento compuesto**, el cual se puede definir como una colección de documentos, posiblemente de muy diferentes tipos (como texto, imágenes, hojas de cálculo, etc.), que se integra de manera similar al nivel de la interfaz de usuario. Una interfaz de usuario que puede manipular documentos compuestos oculta el hecho de que aplicaciones diferentes operan en partes diferentes del documento. Para el usuario, todas las partes se integran de manera similar. Cuando cambiar una parte afecta a otras partes, la interfaz de usuario puede tomar las medidas apropiadas, por ejemplo, mediante la modificación de aplicaciones relevantes.

De manera análoga a la situación descrita en el sistema X Window, las aplicaciones asociadas con un documento compuesto no tienen que ejecutarse en la máquina del cliente. Sin embargo, debiera quedar claro que las interfaces de usuario que soportan documentos compuestos pueden relacionarse más con el proceso que las que no lo hacen.

3.3.2 Software del lado del cliente para transparencia de la distribución

El software del cliente está compuesto por más que solamente interfaces de usuario. En muchos casos, en una aplicación cliente-servidor las partes del proceso y el nivel de datos también se ejecutan del lado del cliente. Una clase especial está formada por software embebido del cliente, digamos como el de cajeros bancarios automáticos (ATM, por sus siglas en inglés), cajas registradoras, lectores de código de barras, cajas TV *set-top*, etc. En estos casos, la interfaz de usuario es una parte relativamente pequeña del software del cliente comparada con el procesamiento local y los medios de comunicación.

Además de la interfaz de usuario y de otro software relacionado con la aplicación, el software del cliente comprende los componentes necesarios para lograr la transparencia de distribución. De manera ideal, un cliente no debiera estar al tanto de que se está comunicando con procesos remotos. Por contraste, la transparencia de distribución es con frecuencia menos transparente para los servidores por razones de rendimiento y de precisión. Por ejemplo, en el capítulo 6 mostraremos que los servidores replicados en ocasiones requieren comunicarse para establecer la manera y el orden en que deben realizarse las operaciones en cada réplica.

Por lo general, la transparencia de acceso es manipulada mediante la generación de una matriz cliente a partir de una definición de interfaz de lo que el servidor tiene que ofrecer. La matriz proporciona la misma interfaz disponible en el servidor, pero oculta las posibles diferencias en las arquitecturas de las máquinas, así como la comunicación real.

Existen diferentes maneras de manipular una ubicación, la migración, y la transparencia de reubicación. El uso de un adecuado sistema de nombres resulta crucial, como veremos en el siguiente capítulo. En muchos casos, también la cooperación con el software del lado del cliente es importante. Por ejemplo, cuando un cliente ya está enlazado a un servidor, puede ser informado directamente cada vez que el servidor cambia de ubicación. En este caso, el middleware del cliente puede ocultar al usuario la ubicación geográfica del servidor, e incluso regresar al servidor si es

necesario. En el peor de los casos, la aplicación del cliente pudiera observar una pérdida temporal de rendimiento.

De manera similar, muchos sistemas distribuidos implementan la transparencia de replicación por medio de soluciones del lado del cliente. Por ejemplo, imagine un sistema distribuido con servidores replicados. Tal replicación se puede lograr mediante el reenvío de una petición a cada réplica, como aparece en la figura 3-10. El software del lado del cliente puede recopilar de manera transparente todas las respuestas y pasar solamente una respuesta a la aplicación del cliente.

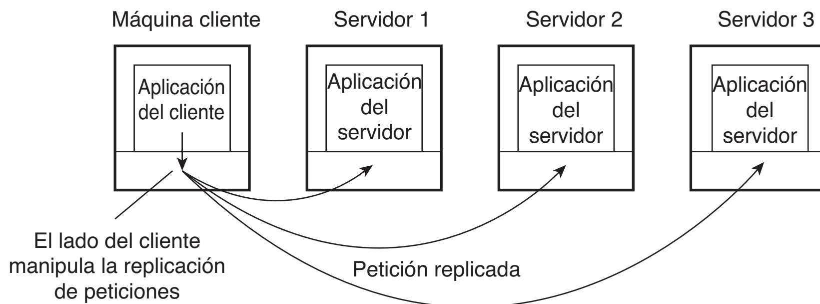


Figura 3-10. Transparencia de replicación de un servidor mediante una solución del lado del cliente.

Por último, consideremos la transparencia a fallas. El enmascaramiento de las fallas de la comunicación con un servidor se hace a través del middleware del cliente. Por ejemplo, podemos configurar el middleware del cliente para intentar de manera repetida la conexión a un servidor, o quizá tratar con otro servidor después de varios intentos. Incluso hay situaciones en las cuales el middleware del cliente devuelve datos que tenía en caché durante una sesión previa, como en ocasiones lo hacen los navegadores web que fallan al intentar conectarse a un servidor.

La transparencia de concurrencia puede manipularse a través de servidores especiales intermedios —monitores para transacciones notables— y requiere menos soporte del software del cliente. De manera similar, a menudo la transparencia de persistencia se manipula por completo del lado del servidor.

3.4 SERVIDORES

Ahora demos un vistazo a la organización de los servidores. En las páginas siguientes, nos concentraremos primero en diversos asuntos relacionados con el diseño general de los servidores, para continuar con la explicación de clústeres de servidores.

3.4.1 Temas generales de diseño

Un servidor es un proceso que implementa un servicio específico en representación de un conjunto de clientes. En esencia, cada servidor está organizado en la misma forma: espera una petición

entrante de un cliente y se asegura de que se haga algo con dicha petición, después de lo cual espera por la siguiente petición entrante.

Existen distintas maneras de organizar servidores. En el caso de un **servidor iterativo**, el propio servidor manipula la petición y, si es necesario, devuelve una respuesta a la petición del cliente. Un **servidor concurrente** no manipula por sí mismo la petición, pero la pasa a un hilo separado o a otro proceso, después de lo cual de inmediato queda en espera de la siguiente petición entrante. Un servidor multihilos es ejemplo de un servidor concurrente. Una implementación alternativa para un servidor concurrente es la de crear un nuevo proceso para cada nueva petición entrante. Este método es seguido en muchos sistemas UNIX. El hilo del proceso que manipula la petición es responsable de devolver una respuesta a la petición del cliente.

Otro problema se presenta cuando los clientes hacen contacto con el servidor. En todos los casos, los clientes envían las peticiones a un **punto final**, también llamado **puerto**, localizado en la máquina donde está corriendo el servidor. Cada servidor atiende un puerto específico. ¿Cómo saben los clientes el puerto de un servicio? Un método es asignar puertos de manera global para servicios muy conocidos. Por ejemplo, los servidores que manipulan las peticiones de internet y FTP siempre atienden al puerto TCP 21. De manera similar, un servidor HTTP para la World Wide Web siempre atenderá al puerto TCP 80. Estos puertos fueron asignados por Internet Assigned Numbers Authority (IANA), y están documentados en Reynolds y Postel (1994). Con los puertos asignados, el cliente solamente necesita encontrar la dirección de red de la máquina en donde corre el servidor. Como explicamos en el siguiente capítulo, los servicios de nombre pueden utilizarse para este propósito.

Existen muchos servicios que no requieren un puerto asignado previamente. Por ejemplo, un servidor de tiempo podría utilizar un puerto que se asigna de manera dinámica mediante su sistema operativo local. En ese caso, un cliente tendrá que buscar primero dicho puerto. Una solución es la de tener un demonio especial activo en cada máquina servidor. El demonio mantiene la ubicación del puerto actualizada mediante cada servicio implementado en el servidor local. El propio demonio atiende a un puerto conocido. Un cliente contactará primero al demonio, solicitará el puerto, y luego contactará al servidor especificado, como ilustra la figura 3-11(a).

Es muy común asociar un puerto con un servicio específico. Sin embargo, la implementación de cada servicio por medio de un servidor separado puede significar en realidad una pérdida de recursos. Por ejemplo, en un sistema UNIX estándar, es común tener muchos servidores en ejecución simultánea, en donde la mayoría espera de manera pasiva hasta que entra una petición del cliente. En lugar de seguir el rastro de tantos procesos pasivos, con frecuencia es más fácil tener un solo **superservidor** que atienda a cada puerto asociado con un servicio específico, como aparece en la figura 3-11(b). Por ejemplo, este método es tomado por el demonio *inetd* en UNIX. *Inetd* atiende cierto número de puertos conocidos para servicios de internet. Cuando entra una petición, el demonio crea un proceso para hacerse cargo de la petición. Dicho proceso culminará una vez terminada la petición.

Otro problema que requiere tomarse en cuenta al diseñar el servidor es si podemos interrumpir un servidor y cómo hacerlo. Por ejemplo, considere un usuario que decide subir un archivo muy grande a un servidor FTP. Después, el usuario repentinamente se da cuenta de que subió el archivo incorrecto, entonces desea interrumpir al servidor para cancelar la transmisión. Existen distintas maneras de hacer esto. Un método que funciona muy bien solamente en internet (y en

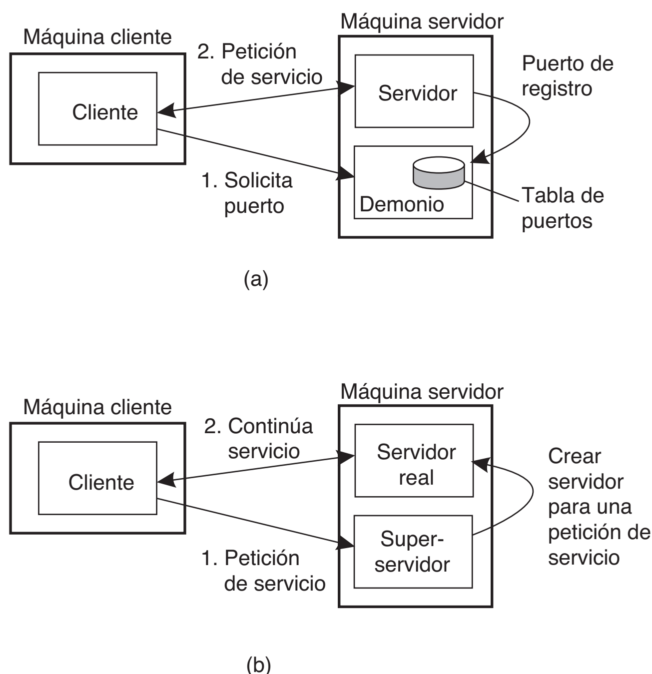


Figura 3-11. (a) Enlace cliente-servidor mediante el uso de un demonio. (b) Enlace cliente-servidor mediante el uso de un superservidor.

ocasiones es la única alternativa) es que el usuario salga abruptamente de la aplicación cliente (lo cual suspenderá la conexión con el servidor), la reinicie de inmediato, y aparente que no sucedió nada. El servidor desechará la conexión anterior suponiendo que probablemente el cliente presentó alguna falla.

Un método mucho mejor para manipular las interrupciones de comunicación es desarrollar al cliente y al servidor de tal modo que sea posible enviar datos **fuera-de-banda**, los cuales son datos a procesar por el servidor antes que cualquier otro dato del cliente. Una solución es dejar que el servidor atienda un puerto de control al cual el cliente envía una petición de datos fuera-de-banda, mientras atiende (con menor prioridad) al puerto a través del cual se envían los datos normales. Otra solución es enviar los datos fuera-de-banda a través de la misma conexión por donde el cliente envía la petición original. Por ejemplo, en TCP es posible transmitir datos urgentes. Cuando el servidor recibe datos urgentes, lo último se interrumpe (por ejemplo, a través de una señal en sistemas UNIX), después de lo cual es posible inspeccionar datos y manipularlos de manera conveniente.

Un problema final e importante del diseño es si el servidor se desplaza o no. Un **servidor sin estado** no mantiene información con respecto al estado de sus clientes, y puede modificar su propio estado sin necesidad de dar información a cliente alguno (Birman, 2005). Por ejemplo, un servidor web es un servidor sin estado. Simplemente requiere de las peticiones HTTP de

entrada, las cuales pueden ser para la carga de un archivo al servidor o (con mayor frecuencia) para recuperar un archivo. Cuando la petición se procesa, el servidor web olvida por completo al cliente. De manera similar, la colección de archivos que el servidor web puede manipular (posiblemente en cooperación con el servidor de archivos) puede cambiar sin que los clientes deban estar informados.

Observe que en muchos sistemas sin estado el servidor realmente mantiene información relacionada con sus clientes, pero resulta crucial que si esta información se pierde no provocará interrupción alguna del servicio que ofrece el servidor. Por ejemplo, generalmente un servidor web registra todas las peticiones del cliente. Por ejemplo, esta información es útil para decidir si un documento se debe replicar, y hacia dónde debiera replicarse. Claramente, no existe otra consecuencia que no sea en la forma de un rendimiento menos óptimo si se pierde dicho registro.

Una forma especial de diseño sin estado es cuando el servidor da mantenimiento a lo que conocemos como **estado suave**. En este caso, el servidor promete mantener el estado que representa al cliente, pero solamente por un tiempo determinado. Una vez que dicho tiempo expira, el servidor de nuevo regresa a su comportamiento habitual, por tanto descarta cualquier información de la cuenta asociada con el cliente. Un ejemplo de este tipo de estado es un servidor implementado para mantener al cliente informado con respecto a actualizaciones, pero solamente por un periodo limitado. Después de eso, al cliente se le solicita buscar en el servidor las actualizaciones. Los métodos de estado suave se originan a partir del diseño de un protocolo en las redes de computadoras, pero se pueden aplicar de igual manera al diseño del servidor (Clark, 1989; y Lui y cols., 2004).

Por el contrario, un **servidor con estados** por lo general mantiene información persistente acerca de sus clientes. Esto significa que el servidor necesita eliminar la información de manera explícita. El ejemplo típico es un servidor de archivos que permite al cliente mantener una copia local de un archivo, incluso para llevar a cabo operaciones de actualización. Tal servidor administra una tabla que contiene entradas (*cliente, archivo*). La tabla permite al servidor mantener la huella del cliente que actualmente contiene los permisos de actualización sobre cierto archivo, y de esta manera también la versión más reciente de dicho archivo.

Este método puede mejorar el rendimiento de las operaciones de lectura y escritura que percibe el cliente. Con frecuencia la mejora en el rendimiento con respecto a los servidores sin estado es un importante beneficio de los diseños con estado. Sin embargo, el ejemplo también muestra la peor desventaja de los servidores con estado. Si el servidor se estropea, tiene que recuperar su tabla de (*cliente, archivo*) entradas, o de lo contrario no puede garantizar que ha procesado las actualizaciones más recientes sobre un archivo. En general, un servidor con estado necesita recuperar su estado completo tal como estaba antes de arruinarse. Como explicaremos en el capítulo 8, habilitar la recuperación puede incluir una considerable complejidad. En el diseño sin estado, no se requiere tomar medidas especiales para recuperar un servidor después de una anomalía. Simplemente comienza de nuevo la ejecución, y espera otra vez las peticiones de los clientes.

Ling y colaboradores (2004) argumentan que en realidad deberíamos distinguir entre un **estado de sesión** (temporal) y un estado permanente. El ejemplo anterior es típico de un estado de sesión: está asociado con una serie de operaciones para un solo usuario y debiera ser mantenido por un tiempo determinado, pero no de manera indefinida. Como resultado, con frecuencia el estado de

sesión es mantenido mediante arquitecturas cliente-servidor de tres capas, en donde el servidor de aplicaciones realmente requiere acceder al servidor de la base de datos a través de una serie de consultas antes de ser capaz de responder al cliente que hace la petición. Aquí el asunto es que no hay daño real alguno si se pierde el estado de una sesión, lo cual permite que el cliente simplemente reenvíe la petición original. Esta observación propicia un almacenamiento de estado más simple y menos confiable.

Lo que permanece como un estado permanente es, por lo general, información mantenida en base de datos, tal como la información de los clientes, llaves asociadas con la compra de software, etc. Sin embargo, para la mayoría de los sistemas distribuidos, mantener el estado de sesión implica un diseño de estados que requiere medidas especiales cuando ocurren fallas y hacer suposiciones explícitas con respecto a la durabilidad del estado almacenado en el servidor. Regresaremos a estos temas de manera más extensa al explicar la tolerancia a fallas.

Al diseñar un servidor, la opción para un diseño sin estado o con estado no debe afectar los servicios proporcionados por el servidor. Por ejemplo, si los archivos se tienen que abrir antes de poder leer o escribir en ellos, entonces un servidor sin estado debe de una forma o de otra imitar este comportamiento. Una solución muy común, la cual explicaremos con más detalle en el capítulo 11, es que el servidor responde a una petición de lectura o escritura al abrir primero el archivo referido, luego realiza la lectura real o las operaciones de escritura, y de inmediato cierra nuevamente el archivo.

En otros casos, un servidor pudiera querer mantener un registro con respecto al comportamiento del cliente de modo que pueda responder de manera más efectiva a sus peticiones. Por ejemplo, en ocasiones los servidores web ofrecen la posibilidad de direccionar de inmediato a un cliente hacia sus páginas favoritas. Cuando el servidor no puede mantener el estado, entonces una solución común es dejar que el cliente envíe también información adicional relacionada con sus accesos previos. En el caso de la web, esta información con frecuencia es almacenada de manera transparente por el navegador del cliente en lo que conocemos como una **cookie**, la cual es una pequeña pieza de datos con información específica del cliente que es de interés para el servidor. Las cookies nunca son ejecutadas por el servidor, solamente almacenadas.

La primera vez que un cliente accede a un servidor, éste envía una cookie junto con las páginas web requeridas de nuevo hacia el navegador, después de lo cual el navegador retira la cookie. Cada vez posterior que el cliente accede al servidor, se envía la cookie para dicho servidor junto con la petición. Aunque en principio este método funciona bien, el que el navegador envíe las cookies de vuelta para su salvaguarda con frecuencia se oculta por completo a los usuarios. Mucho por privacidad. A diferencia de la mayoría de las cookies de la abuela, éstas deben permanecer en el lugar donde fueron cocinadas.

3.4.2 Servidores de clústeres

En el capítulo 1 explicamos brevemente la computación en cluster como una de las muchas apariencias de los sistemas distribuidos. Ahora daremos un vistazo a la organización de los servidores de clústeres, junto con los problemas que surgen en su diseño.

Organización general

Visto de manera simple, un servidor de cluster no es otra cosa que una colección de máquinas conectadas a través de una red, donde cada máquina ejecuta uno o más servidores. Los servidores de clústeres que consideramos aquí son aquellos en los cuales las máquinas están conectadas mediante una red de área local, con frecuencia ofreciendo un gran ancho de banda y latencia muy pequeña.

En la mayoría de los casos, un servidor de cluster está organizado dentro de tres niveles, como podemos ver en la figura 3-12. El primer nivel consta de un interruptor (switch) lógico a través del cual se rutean las peticiones del cliente. Tal interruptor puede variar ampliamente. Por ejemplo, los interruptores ubicados en la capa de transporte aceptan peticiones de conexión TCP entrantes y pasan dichas peticiones a uno de los servidores incluidos en el cluster, como explicaremos más adelante. Un ejemplo completamente diferente es un servidor web que acepta peticiones HTTP entrantes, pero que permite el paso parcial de peticiones a los servidores de aplicación para un proceso posterior solamente para después recopilar los resultados y devolver una respuesta HTTP.

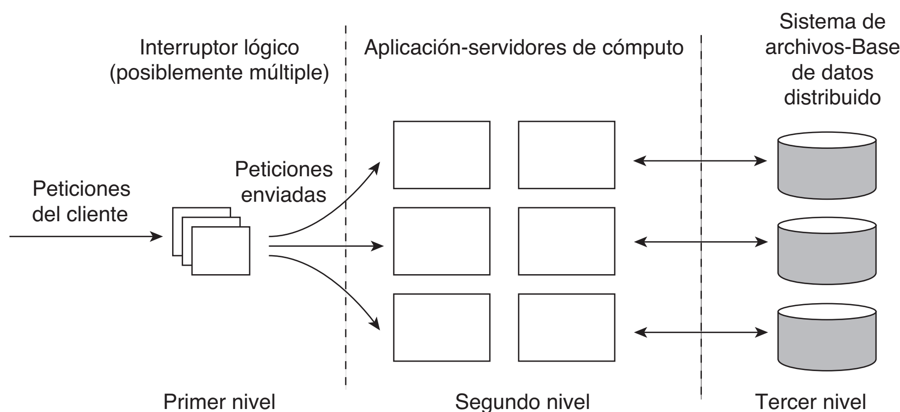


Figura 3-12. Organización general del cluster de servidores de tres niveles.

Como en cualquier arquitectura cliente-servidor multiniveles, muchos servidores de clústeres también contienen servidores dedicados al proceso de las aplicaciones. En el cómputo en cluster, por lo general, son servidores que ejecutan hardware de alto rendimiento dedicado a entregar poder de cómputo. Sin embargo, en el caso de servidores de clústeres empresariales, podría darse el caso de que las aplicaciones requieran ejecutarse en máquinas de este tipo (low-end) en las cuales el poder de cómputo requerido no es el cuello de botella, pero el acceso al almacenamiento sí lo es.

Esto nos da el tercer nivel, el cual consiste en servidores de procesamiento de datos, archivos notables, y servidores de bases de datos. De nuevo, dependiendo del uso del cluster de servidores, dichos servidores pudieran ejecutarse en máquinas especializadas, configuradas para acceso a discos de alta velocidad y que cuenten con grandes cachés de datos del lado del servidor.

Por supuesto, no todos los clústeres de servidor seguirán esta estricta separación. Es frecuente que cada máquina esté equipada con su propio almacenamiento global, a menudo la integración de la aplicación y del proceso de datos dentro de un solo servidor provocan una arquitectura de dos niveles. Por ejemplo, cuando tratamos con flujos de medios utilizando un servidor de clústeres es común la instalación de una arquitectura de dos niveles, en donde cada máquina actúa como un servidor de medios dedicado (Steinmetz y Nahrstedt, 2004).

Cuando un servidor de clústeres ofrece múltiples servicios, pudiera suceder que máquinas diferentes ejecuten servidores de aplicaciones distintos. En consecuencia, el switch tendrá que ser capaz de distinguir los servicios, de lo contrario no podrá reenviar las peticiones a las máquinas apropiadas. Como resultado, muchas máquinas de dos niveles ejecutan solamente una sola aplicación. Esta limitación surge de la dependencia entre el software disponible y el hardware, pero además porque a menudo aplicaciones diferentes son ejecutadas por administradores diferentes. Lo anterior implica el no interferir con cada una de las otras máquinas.

En consecuencia, pudiéramos encontrar que ciertas máquinas están temporalmente ociosas (en espera), mientras que otras reciben una sobrecarga de peticiones. Lo que podría ser útil es migrar de manera temporal los servicios hacia las máquinas ociosas. Una solución propuesta en Awadallah y Rosenblum (2004) es utilizar máquinas virtuales, ello permite una migración relativamente fácil de código hacia las máquinas reales. Regresaremos a la migración de código más adelante en este capítulo.

Demos un vistazo más cercano al primer nivel, que consta de un switch. Una meta importante en el diseño de servidores de clústeres es ocultar el hecho de que existen múltiples servidores. En otras palabras, las aplicaciones cliente que corren en máquinas remotas no deben tener necesidad de saber cosa alguna acerca de la organización interna de un cluster. Esta transparencia de acceso se ofrece invariablemente por medio de un solo punto de acceso, en cambio se implementa a través de algún tipo de interruptor de hardware tal como una máquina dedicada.

El interruptor forma un punto de entrada para el cluster del servidor, lo que ofrece una sola dirección de red. Por escalabilidad y disponibilidad, un servidor de cluster pudiera tener múltiples puntos de acceso, donde cada punto de acceso se crea utilizando una máquina dedicada por separado. Solamente consideraremos el caso de un solo punto de acceso.

Una forma estándar de acceder al servidor de cluster es configurando una conexión TCP sobre la que las peticiones al nivel de la aplicación se envíen como parte de la sesión. Una sesión termina al interrumpir la conexión. En el caso de **los switches de la capa de transporte**, el switch acepta las peticiones entrantes de la conexión TCP y direcciona las conexiones hacia uno de los servidores (Hunt y cols., 1997; y Pai y cols., 1998). En la figura 3-13 mostramos el principio de funcionamiento de lo que llamamos comúnmente **TCP handoff**.

Cuando el interruptor recibe una petición de conexión TCP, identifica de manera subsecuente al mejor servidor disponible para manejar la petición, y reenvía el paquete de la petición a dicho servidor. El servidor, a su vez, enviará un reconocimiento de vuelta al cliente que hace la petición, pero insertando la dirección IP del interruptor como el campo fuente del encabezado de paquete IP que acarrea el segmento TCP. Observe que este reconocimiento es necesario para que el cliente pueda continuar la ejecución del protocolo TCP: está esperando una respuesta por parte del switch, no de algún servidor aleatorio del cual nunca ha oído hablar. Claramente, una implementación de TCP handoff requiere modificaciones al nivel del sistema operativo.

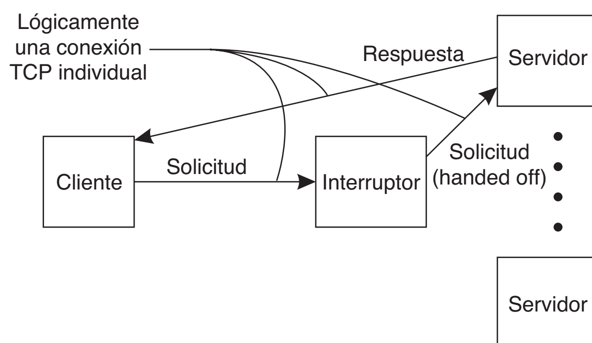


Figura 3-13. El principio del TCP handoff.

Podemos advertir en este punto que el switch puede jugar un papel muy importante en la distribución de la carga a lo largo de distintos servidores. Al decidir hacia dónde reenviar una petición, el switch decide también qué servidor debe manipular el proceso subsecuente de dicha petición. La política más sencilla de balanceo de carga que puede seguir el switch es la de round-robin: cada vez toma el siguiente servidor de su lista para reenviar una petición.

También podemos instalar criterios de selección del servidor más avanzados. Por ejemplo, asumamos que el servidor de cluster ofrece múltiples servicios. Si el switch puede distinguir entre dichos servicios cuando entra una petición, entonces puede tomar decisiones informadas acerca del lugar al cual enviar la petición. Esta selección de servidor puede tener lugar aún al nivel de transporte, los servicios proporcionados se distinguen por medio de un número de puerto. Un paso adelante es hacer que el switch inspeccione realmente el costo de la petición entrante. Este método se puede aplicar solamente cuando se sabe cuál pudiera ser el costo. Por ejemplo, en el caso de servidores web, el switch puede esperar en algún momento una petición HTTP, con base en ella puede decidir quién procesará la petición. Regresaremos a dicha **distribución de peticiones basada en su contenido** cuando expliquemos los sistemas basados en web en el capítulo 12.

Servidores distribuidos

Los servidores de clústeres explicados hasta este momento son configurados generalmente de manera estática. En estos clústeres, con frecuencia existe una máquina separada para efectuar la administración que mantiene la huella de los servidores disponibles, y pasa esta información a otras máquinas conforme sea apropiado, tales como el switch.

Como ya mencionamos, la mayoría de los clústeres de información ofrecen un solo punto de acceso. Cuando este punto falla, el cluster se vuelve no disponible. Para eliminar este potencial problema podemos proporcionar varios puntos de acceso, de los cuales se hacen públicas las direcciones. Por ejemplo, el **Servicio de Nombres de Dominio** (**DNS**, por sus siglas en inglés) puede devolver diversas direcciones, todas pertenecientes al mismo nombre del servidor. Este método también requiere que los clientes realicen varios intentos si uno de los servidores falla. Más aún, esto no resuelve el problema de requerimiento de puntos de acceso estáticos.

Tener estabilidad, tal como un punto de acceso con vida prolongada, es una característica deseable desde la perspectiva de los clientes y de los servidores. Por otro lado, también es deseable tener un alto grado de flexibilidad en la configuración de un servidor de cluster, incluyendo al switch. Esta observación conduce al diseño de un **servidor distribuido** el cual, en efecto, posiblemente no es nada más que un conjunto de máquinas que cambian de manera dinámica, y posiblemente también con más de un punto de acceso, pero que sin embargo, aparecen ante el mundo como una sola y poderosa máquina. El diseño de dicho servidor distribuido está dado por Szymaniak y colaboradores (2005), y lo explicaremos brevemente más adelante.

La idea básica detrás de un servidor distribuido es que los clientes se beneficien a partir de un servidor fuerte, estable, y de alto rendimiento. Con frecuencia estas propiedades puede proporcionarlas una mainframe de alto nivel, de las cuales algunas tienen una aclamada reputación lograda a lo largo de más de 40 años de servicio. Sin embargo, al agrupar múltiples máquinas más simples de manera transparente dentro de un cluster, y no confiar en la disponibilidad de una sola máquina, podría ser posible lograr un mejor grado de estabilidad que con cada componente de manera individual. Por ejemplo, dicho cluster pudiera ser configurado en forma dinámica desde las máquinas del usuario, como en el caso de un sistema distribuido de colaboración.

Concentrémonos en la manera en que un punto de acceso estable se puede lograr en dicho sistema. La idea principal es hacer uso de servicios de red disponibles, un soporte importante para la IP versión 6 (MIPv6). En MIPv6, asumimos un nodo móvil para tener una **red doméstica** en donde resida normalmente, y para la cual tiene una dirección estable asociada conocida como **dirección doméstica (HoA)**. Esta red doméstica tiene un ruteador especial adjunto, conocido como **agente doméstico**, el cual cuidará del tráfico hacia el nodo móvil cuando se encuentre lejos. Hasta este punto, cuando un nodo móvil se adjunta a una red externa, recibirá una **dirección añadida cuidadosamente** (**CoA**) en donde puede ser localizado. Esta dirección añadida cuidadosamente es reportada al agente doméstico del nodo, el cual verá entonces que todo el tráfico se reenvía al nodo móvil. Observe que las aplicaciones que se comunican con el nodo móvil solamente verán la dirección asociada con la red del nodo doméstico. Nunca verán la dirección añadida cuidadosamente.

Este principio se puede utilizar para ofrecer una dirección estable de un servidor distribuido. En este caso, una sola **dirección de contacto** se asigna inicialmente al cluster de servidores. La dirección de contacto será la dirección de toda la vida del servidor para que se use en toda comunicación con el mundo exterior. En cualquier momento, un nodo implementado en el servidor distribuido operará como un punto de acceso mediante el uso de dicha dirección de contacto, pero este rol lo puede tomar fácilmente otro nodo. Lo que sucede es que el punto de acceso registra su propia dirección como la dirección añadida cuidadosamente con el agente doméstico asociado con el servidor distribuido. En ese punto, todo el tráfico será direccionado hacia el punto de acceso, el cual cuidará de distribuir las peticiones a lo largo de los nodos que actualmente participan. Si el punto de acceso falla, un simple mecanismo de falla entra en su lugar, por medio del cual otro punto de acceso comunica una nueva dirección añadida cuidadosamente.

Esta sencilla configuración haría del agente doméstico, así como del punto de acceso, un potencial cuello de botella mientras todo el tráfico fluya por estas dos máquinas. Esta situación puede evitarse al utilizar el método MIPv6 conocido como *optimización de ruta*. La optimización de ruta trabaja de la siguiente manera. Cada vez que un nodo móvil con dirección doméstica *HA* informa

su dirección añadida cuidadosamente actual, digamos *CA*, el agente doméstico puede reenviar *CA* a un cliente. Éste almacenará de manera local el par (*HA, CA*). A partir de ese momento en adelante, la comunicación será reenviada a *CA*. Aunque en el lado del cliente la aplicación todavía puede utilizar la dirección doméstica, el software de soporte subyacente para MIPv6 traducirá dicha dirección a *CA* y utilizará esta última en su lugar.

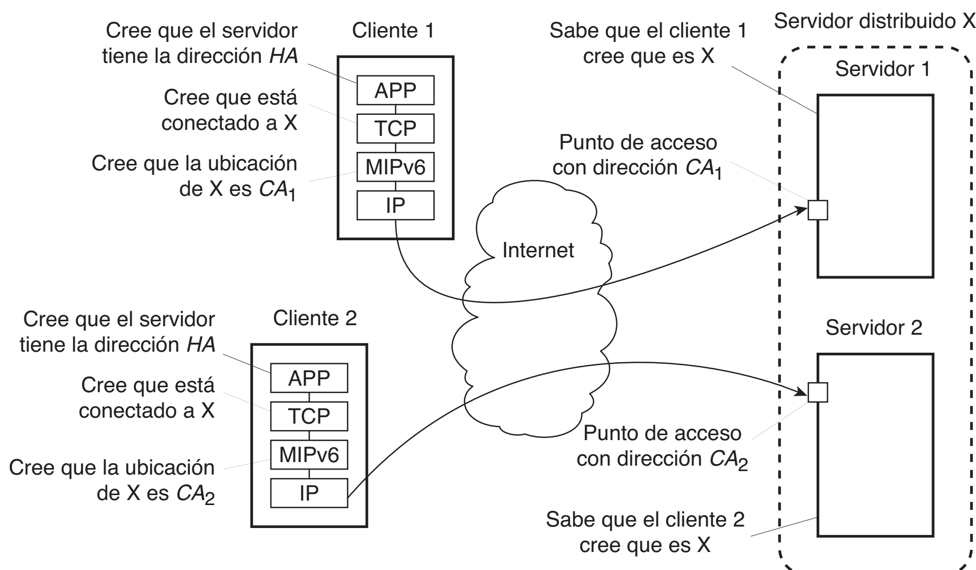


Figura 3-14. Optimización de ruta en un servidor distribuido.

La optimización de ruta se puede utilizar para hacer que diferentes clientes crean que se comunican con un solo servidor, donde, de hecho, cada cliente se comunica con un nodo miembro diferente del servidor distribuido, tal como ilustra la figura 3-14. Hasta aquí, cuando un punto de acceso de un servidor distribuido reenvía una petición del cliente C_1 a, digamos el nodo S_1 (con dirección CA_1), pasa suficiente información hacia S_1 para dejarla iniciar el procedimiento de optimización de ruta mediante el cual en algún momento se le hace creer al cliente que la dirección añadida cuidadosamente es CA_1. Esto permitirá a C_1 almacenar el par (HA, CA_1). Durante este proceso, el punto de acceso (así como el agente doméstico) dirige la mayor parte del tráfico entre C_1 y S_1. Esto previene que el agente doméstico crea que la dirección añadida cuidadosamente ha cambiado, de modo que continuará comunicándose con el punto de acceso.

Por supuesto, mientras este procedimiento de optimización de ruta tiene lugar, las peticiones de otros clientes aún pueden entrar. Éstas permanecen en un estado pendiente en el punto de acceso hasta que puedan ser reenviadas. La petición de otro cliente C_2 puede entonces reenviarse a un nodo miembro S_2 (con dirección CA_2), ello permite que tal petición deje al cliente C_2 almacenar el

punto (HA, CA_2). Como resultado, clientes diferentes se comunicarán de manera directa con diferentes miembros del servidor distribuido, donde cada aplicación cliente aún tiene la ilusión de que este servidor tiene la dirección HA. El agente doméstico continúa la comunicación con el punto de acceso comunicándose con la dirección de contacto.

3.4.3 Administración de servidores de clústeres

Un servidor de cluster debe aparecer ante el mundo exterior como una sola computadora, como en realidad es frecuentemente el caso. Sin embargo, cuando se trata de dar mantenimiento al cluster, la situación cambia de manera dramática. Se han hecho diversos intentos para facilitar la administración de un servidor de cluster, según veremos a continuación.

Métodos comunes

Por mucho, el método más común para administrar un servidor de cluster es extender las funciones tradicionales de administración de una sola computadora al cluster. En su modo más primitivo, esto significa que un administrador puede acceder a un nodo desde un cliente remoto y ejecutar comandos locales de administración para monitorear, instalar, y modificar los componentes.

Algo más avanzado es ocultar el hecho de que se necesita acceder a un nodo y en lugar de eso proporcionar una interfaz a una máquina para la administración que permita recopilar la información desde uno o más servidores, actualizar sus componentes, agregar y remover nodos, etc. La ventaja principal del último método es que las operaciones colectivas, que operan en un grupo de servidores, pueden proporcionarse de manera más fácil. Este tipo de administración de servidores de clústeres se aplica extensamente en la práctica, y se puede ejemplificar mediante un software de administración como Cluster Systems Management de IBM (Hochstetler y Beringer, 2004).

Sin embargo, tan pronto como el cluster crece más allá de varias decenas de nodos, este tipo de administración ya no es el más apropiado. Muchos centros de datos requieren la administración de cientos de servidores, organizados en muchos clústeres pero operando de manera colaborativa. Hacer esto por medio de una administración centralizada de servidores simplemente está fuera de contexto. Más aún, fácilmente podemos advertir que los clústeres muy grandes requieren una administración continua de reparaciones (incluyendo las actualizaciones). Para simplificar las tareas, si p es la probabilidad de que un servidor esté fallando actualmente, y asumimos que las fallas son independientes, entonces para un cluster de N servidores operar sin que haya un solo servidor con falla es $(1 - p)^N$. Con $p = 0.001$ y $N = 1\,000$, solamente existe un 36 por ciento de oportunidad de que todos los servidores funcionen correctamente.

Como podemos apreciar, el soporte para grandes servidores de clústeres por lo general es el adecuado. Existen varias reglas de oro que se deben considerar (Brewer, 2001), pero no existe un método sistemático para tratar con la administración de sistemas masivos. La administración de clústeres es todavía incipiente, aunque podemos esperar que las soluciones de autoadministración que explicamos en el capítulo anterior encontrarán su camino en esta área, cuando tengamos más experiencia en ellas.

Ejemplo: PlanetLab

Ahora demos un vistazo a un servidor de cluster que de alguna manera es inusual. PlanetLab es un sistema distribuido y colaborativo en el cual diversas empresas donan una o más computadoras, totalizando miles de nodos. Juntas, estas computadoras forman un cluster de un nivel donde acceso, procesamiento, y almacenamiento pueden tener lugar de manera individual en cada nodo. La administración PlanetLab es, por necesidad, en su mayor parte distribuida. Antes de explicar sus principios básicos, describamos las principales características de su arquitectura (Peterson y cols., 2005).

En PlanetLab, una empresa dona uno o más nodos, donde cada nodo es una simple computadora, aunque pudiera ser además un cluster de máquinas por sí solo. Cada nodo está organizado como vemos en la figura 3-15. Existen dos componentes importantes (Bavier y cols., 2004). La primera es el monitor de la máquina virtual (VMM), que es un sistema operativo Linux mejorado. Las mejoras comprenden primordialmente ajustes para el soporte del segundo componente, denominado, **vservers**. Podemos imaginar un vserver (Linux) como un ambiente independiente en el cual se ejecuta un conjunto de procesos. Los procesos de los diferentes vservers son *completamente* independientes. No pueden compartir de manera directa ningún recurso tal como archivos, memoria principal, ni conexiones de red como es lo normal con procesos en ejecución en la parte superior de los sistemas operativos. En lugar de ello, los vservers proporcionan un ambiente que consta de su propia colección de paquetes de software, programas, e instalaciones de red. Por ejemplo, un vserver pudiera proporcionar un ambiente en el cual un proceso advierte que puede hacer uso de Phyton 1.5.2 en combinación con un servidor web Apache más antiguo, digamos *httpd 1.3.1*. Por contraste, otro vserver pudiera soportar la última versión de Phyton y *httpd*. En este sentido, llamar al vserver "servidor" es un poco erróneo ya que en realidad solamente aisla conjuntos de procesos entre sí. Más adelante regresaremos al tema de los vservers.

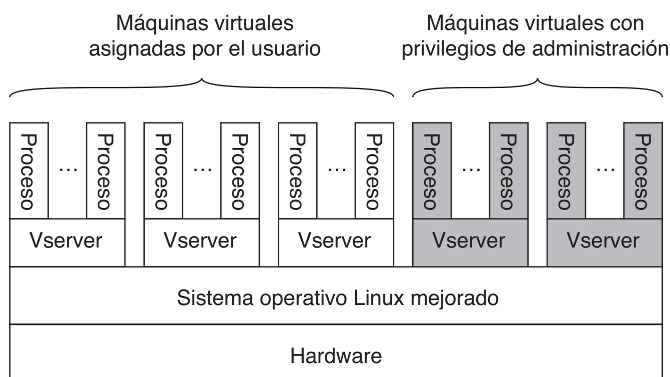


Figura 3-15. Organización básica de un nodo PlanetLab.

El VMM de Linux asegura que los vservers se encuentren separados: los procesos residentes en diferentes vservers se ejecutan de manera concurrente e independiente, cada uno sólo usa paquetes

de software y programas disponibles en su propio ambiente. El aislamiento entre procesos residentes en diferentes vservers es estricto. Por ejemplo, dos procesos ejecutándose en diferentes vservers pueden tener el mismo ID de usuario, pero esto no implica que parten del mismo usuario. Esta separación facilita considerablemente el soporte a los usuarios desde organizaciones diferentes que desean utilizar PlanetLab para, por ejemplo, un ambiente de pruebas donde experimentar con sistemas distribuidos y aplicaciones completamente distintas.

Para dar soporte a dichos experimentos, PlanetLab introduce el concepto de un **slice** (partición), el cual es un conjunto de vservers, cada vserver se ejecuta en un nodo diferente. De esta manera, una partición puede imaginarse como un cluster de servidores virtuales implementado por medio de una colección de máquinas virtuales. En PlanetLab, las máquinas virtuales se ejecutan en la parte superior del sistema operativo Linux, el cual se ha extendido mediante cierto número de módulos de kernel.

Existen diversos aspectos que hacen de PlanetLab un problema muy especial. Los tres principales son:

1. Los nodos pertenecen a diferentes organizaciones. A cada organización debiera permitírsele la ejecución de aplicaciones en sus nodos, y restringir el uso de recursos de manera apropiada.

2. Existen distintas herramientas de monitoreo disponibles, pero todas asumen una combinación muy específica de hardware y software. Más aún, todas están adaptadas para usarse dentro de una sola organización.

3. Los programas para diferentes particiones pero ejecutados en el mismo nodo no deben interferir entre sí. Este problema es similar a la independencia de procesos en los sistemas operativos.

Analicemos con más detalle cada uno de estos problemas.

La central para administrar los recursos de PlanetLab es al **administrador de nodos**. Cada nodo cuenta con dicho administrador, implementado por medio de un vserver separado, cuya única tarea es crear otros vservers en el nodo que administra y controlar la ubicación de los recursos. El administrador de nodos no toma decisión alguna con respecto a las políticas; es solamente un mecanismo que proporciona los ingredientes esenciales para ejecutar un programa dentro de un nodo establecido.

El mantenimiento de un registro se lleva a cabo por medio de una especificación de recursos, o *rspec* para abreviar. Una *rspec* especifica un intervalo de tiempo durante el cual se ubican ciertos recursos. Los recursos incluyen espacio en disco, descriptores de archivos, ancho de banda interno y externo, puntos finales a nivel de transporte, memoria principal, y el uso de la CPU. Una *rspec* se identifica a través de un identificador único y global de 128 bits conocido como una capacidad de recurso (*rcap*). Dada una *rcap*, al administrador de nodos puede buscar la *rspec* asociada dentro de una tabla local.

Los recursos están ligados a las particiones. En otras palabras, con objeto de utilizar los recursos, es necesario crear una partición. Cada partición se asocia con un **proveedor de servicio**, al cual podemos interpretar mejor como una entidad que tiene una cuenta en PlanetLab. Entonces, cada

partición se puede identificar mediante un par (*id_principal*, *etiqueta_corte*), en donde *id_principal* identifica al proveedor y *etiqueta_corte* es un identificador que elige el proveedor.

Para crear una nueva partición, cada nodo ejecutará un **servicio de creación de partición** (**SCS**, por sus siglas en inglés), el cual, en cambio, puede contactar al administrador de nodos para solicitar la creación de un vserver y alojar los recursos. No es posible contactar al administrador de nodos por sí solo de manera directa sobre la red, ello le permite concentrarse solamente en la administración local de un recurso. En cambio, el SCS no aceptará las peticiones para creación de particiones de cualquiera. Solamente **autoridades de particiones** específicas son elegibles para solicitar la creación de una partición. Cada autoridad de partición tendrá derechos de acceso a una colección de nodos. El modelo más sencillo es cuando existe solamente una autoridad de partición a la que se le permite solicitar la creación de una partición en todos los nodos.

Para completar el panorama, un proveedor de servicios hará contacto con una autoridad de partición y solicitará la creación de una partición a través de una colección de nodos. Por ejemplo, el proveedor de servicios será reconocido por la autoridad de partición debido a que previamente fue autentificado y posteriormente registrado como un usuario de PlanetLab. En la práctica, los usuarios de PlanetLab contactan a una autoridad de partición por medio de un servicio basado en web. Podemos conocer detalles adicionales en Chun y Spalink (2003).

Lo que revela este procedimiento es que la administración de PlanetLab se realiza a través de intermediarios. Una clase importante de tales intermediarios está formada por las autoridades de partición. Dichas autoridades obtuvieron sus credenciales para crear particiones desde los nodos. La obtención de estas credenciales se logró fuera de los límites, en esencia mediante el contacto con los administradores del sistema en varios sitios. Desde luego, éste es un proceso que consume un tiempo que no invierten los usuarios finales (o, en la terminología de PlanetLab, proveedores del servicio).

Además de autoridades de particiones, hay también autoridades de administración. Donde la autoridad de partición se concentra solamente en la administración de partición, una autoridad de administración es responsable de vigilar los nodos. En especial, asegurar que los nodos bajo su régimen ejecuten el software básico de PlanetLab y se respeten las reglas establecidas por PlanetLab. Los proveedores de servicio confían en que la autoridad de administración proporcionará nodos que se comportarán de manera apropiada.

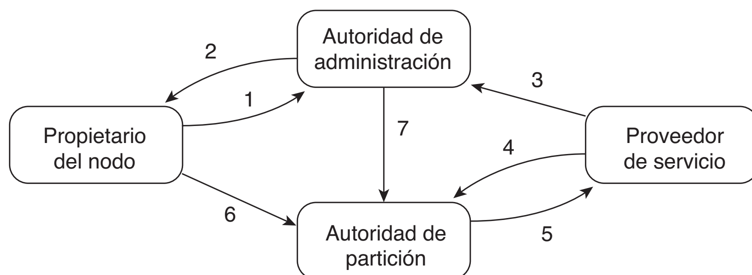


Figura 3-16. Relaciones de administración entre las distintas entidades de PlanetLab.

Esta organización propicia la estructura de administración que aparece en la figura 3-16, descrita en términos de relaciones de confianza en Peterson y colaboradores, (2005). Las relaciones son las siguientes:

1. El propietario de un nodo coloca su nodo dentro del régimen de una autoridad de administración, y posiblemente restringe el uso en donde sea apropiado.
2. Una autoridad de administración proporciona el software necesario para agregar un nodo a PlanetLab.
3. Un proveedor de servicio se registra a sí mismo con una autoridad de administración, y le confía el suministro de nodos que se comportan de manera apropiada.
4. Un proveedor de servicio contacta a una autoridad de partición para crear una partición en una colección de nodos.
5. La autoridad de partición requiere la autentificación de un proveedor de servicio.
6. Un propietario de nodo proporciona un servicio de creación de partición para una autoridad de partición para crear particiones. Básicamente delega la administración de recursos a la autoridad de partición.
7. Una autoridad de administración delega la creación de particiones a una autoridad de partición.

Estas relaciones cubren el problema de la delegación de nodos en forma controlada de modo que el propietario de un nodo se apoye en una administración honesta y segura. El segundo problema que requiere manejarse es el monitoreo. Se necesita un método unificado para permitir a los usuarios ver qué tan bien se comportan sus programas dentro de una partición específica.

PlanetLab lleva a cabo un sencillo método. Cada nodo está equipado con una serie de monitores, cada monitor es capaz de enviar información tal como el uso de la CPU, actividad del disco, entre otros aspectos. Los monitores pueden ser arbitrariamente complejos, pero el problema importante es que siempre envían información sobre la base de nodo por nodo. Esta información se vuelve disponible por medio de un servidor web: cada monitor es accesible a través de sencillas peticiones HTTP (Bavier y cols., 2004).

Hay que admitirlo, este método de monitoreo es aún muy primitivo, pero debiera verse como una base para esquemas avanzados de monitoreo. Por ejemplo, no existe, en principio, razón alguna por la cual Astrolabe, el cual explicamos en el capítulo 2, no se pueda usar para las lecturas agregadas de los monitores a lo largo de múltiples nodos.

Por último, para llegar a nuestro tercer problema de administración, digamos la protección de los programas entre sí, PlanetLab utiliza servidores virtuales de Linux (llamados vservers) para aislar las particiones. Como ya mencionamos, la idea principal de un vserver es ejecutar aplicaciones en su propio ambiente, lo cual incluye todos los archivos que por lo general están compartidos a lo largo de una sola máquina. Tal separación se puede lograr de manera relativamente fácil por medio del comando UNIX chroot, el cual modifica de manera efectiva la raíz del sistema de archivos desde la cual las aplicaciones buscarán archivos. Solamente el superusuario puede ejecutar chroot.

Por supuesto, necesitamos más. Los servidores virtuales Linux no solamente separan el sistema de archivos, sino que por lo general comparten información sobre procesos, direcciones de red, uso de memoria, y así por el estilo. En consecuencia, una máquina física en realidad se particiona en diversas unidades, cada unidad corresponde a un completo y totalmente funcional ambiente Linux aislado de las demás partes. Podemos encontrar un resumen de los servidores virtuales Linux en Potzl y colaboradores, (2005).

3.5 MIGRACIÓN DE CÓDIGO

Hasta aquí, nos hemos preocupado primordialmente de los sistemas distribuidos en los cuales la comunicación está limitada para el paso de datos. Sin embargo, existen situaciones en las cuales el paso de programas, a veces incluso mientras se ejecutan, simplifica el diseño de un sistema distribuido. En esta sección daremos un vistazo cercano a lo que realmente es la migración de código. Iniciaremos mediante la consideración de diferentes métodos para implementar la migración de código, seguida por una explicación con respecto a la manera de tratar con los recursos locales que utiliza un programa en migración. Un problema particularmente difícil es la migración de código en sistemas heterogéneos, la cual también explicaremos.

3.5.1 Métodos para la migración de código

Antes de revisar las diferentes formas de migración de código, consideremos primero el porqué migrar código pudiera ser útil.

Razones para la migración de código

De manera tradicional, la migración de código en sistemas distribuidos se realizó en la forma de **migración de procesos** en los cuales todo un proceso se trasladaba de una máquina a otra (Milojicic y cols., 2000). El traslado de un proceso en ejecución hacia una máquina diferente es una tarea costosa e intrincada, y debiera existir una mejor razón para hacerlo. Esta razón siempre ha sido el rendimiento. La idea básica es que se puede mejorar el rendimiento general del sistema si los procesos se trasladan desde una máquina muy saturada hacia máquinas ligeramente saturadas. Con frecuencia, la carga se expresa en términos de la longitud de la cola o de la inicialización de la CPU, pero también se utilizan otros indicadores de rendimiento.

Los algoritmos de distribución mediante los cuales tomamos las decisiones con respecto a la ubicación y distribución de tareas referentes a un conjunto de procesadores juegan un papel importante en los sistemas de cómputo intenso. Sin embargo, en muchos sistemas distribuidos modernos, la optimización de la capacidad de cómputo es menos importante que, por ejemplo, tratar de minimizar la comunicación. Más aún, debido a la heterogeneidad de las plataformas subyacentes y de las redes de computadores, la mejora en el rendimiento a través de la migración de código se basa generalmente en razones cualitativas en lugar de en modelos matemáticos.

Como ejemplo, considere un sistema cliente-servidor en donde el servidor administra una base de datos de gran tamaño. Si el cliente de una aplicación necesita realizar muchas operaciones de

bases de datos que involucran grandes cantidades de datos, pudiera ser mejor enviar parte de la aplicación del cliente hacia el servidor y enviar por la red solamente los resultados. De lo contrario, la red podría colapsarse con la transferencia de datos desde el servidor hacia el cliente. En este caso, la migración de código se basa en la presunción de que a menudo tiene sentido procesar los datos cerca de donde residen.

Podemos utilizar el mismo razonamiento para migrar partes del servidor al cliente. Por ejemplo, en muchas aplicaciones interactivas de base de datos, los clientes necesitan llenar formas que posteriormente se traducen a series de operaciones en la base de datos. El procesamiento de la forma del lado del cliente, y el sólo envío de la forma completa al servidor, puede en ocasiones evitar que un número relativamente alto de mensajes cortos necesite cruzar la red. El resultado es que el cliente percibe un mejor rendimiento, mientras que el servidor invierte menos tiempo en el procesamiento de la forma y la comunicación.

El soporte para migración de código también puede ayudar a mejorar el rendimiento mediante la explotación del paralelismo, pero sin las usuales e intrincadas relaciones de la programación en paralelo. Un ejemplo típico es la búsqueda de información en la web. Es relativamente sencillo implementar una consulta de búsqueda en la forma de un pequeño programa móvil, llamado **agente móvil**, que se mueve de sitio en sitio. Al hacer distintas copias de dicho programa, y enviar cada copia a diferentes sitios, podemos lograr un aumento lineal de la velocidad para utilizar solamente una instancia individual del programa.

Además de mejorar el rendimiento, existen otras razones para soportar la migración de código. La más importante es la relacionada con la flexibilidad. El método tradicional para construir aplicaciones distribuidas es crear particiones de la aplicación en diferentes partes, y decidir por adelantado el lugar donde se deben ejecutar. Por ejemplo, este método ha desembocado en diferentes aplicaciones cliente-servidor multiniveles, las cuales explicamos en el capítulo 2.

Sin embargo, si el código se puede trasladar entre diferentes máquinas, es posible configurar sistemas distribuidos de manera dinámica. Por ejemplo, suponga que un servidor implementa una interfaz estándar para un sistema de archivos. Para permitir el acceso de clientes remotos al sistema de archivos, el servidor hace uso del protocolo propietario. Por lo general, la implementación del lado del cliente de la interfaz del sistema, basada en dicho protocolo, necesitará estar ligada con la aplicación del cliente. Este método requiere que el software se encuentre disponible para el cliente en el momento que se desarrolla la aplicación del cliente.

Una alternativa es permitir que el servidor proporcione la implementación del cliente no antes de que sea estrictamente necesario, esto es, cuando el cliente se conecta con el servidor. En ese punto, el cliente descarga de manera dinámica la implementación, sigue los pasos necesarios de implementación, y luego invoca al servidor. Este principio aparece en la figura 3-17. Este modelo de código que se traslada dinámicamente desde un sitio remoto requiere que el protocolo para la descarga y la inicialización del código sea estandarizado. Además, es necesario que el código descargado se pueda ejecutar en la máquina del cliente. En los siguientes capítulos explicaremos diferentes soluciones.

La ventaja más importante de este modelo de descarga dinámica del software del cliente es que los clientes no necesitan tener instalado todo el software previamente para poder comunicarse con los servidores. En cambio, se puede introducir el software si es necesario, y de manera similar descar-

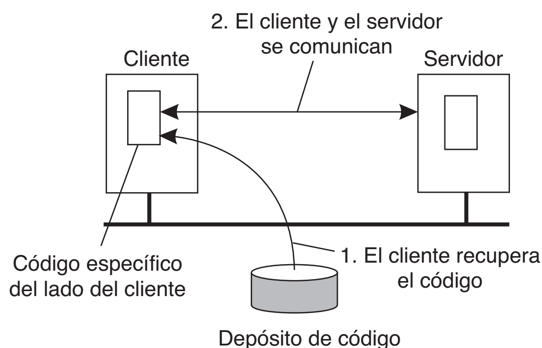


Figura 3-17. El principio de la configuración dinámica de un cliente para comunicarse con un servidor. El cliente primero recupera el software necesario, y luego invoca al servidor.

tarlo cuando ya no se requiera. Otra ventaja es que mientras las interfaces sean estandarizadas, podemos modificar el protocolo del cliente y su implementación con tanta frecuencia como deseemos. Los cambios no afectarán las aplicaciones existentes en la máquina del cliente y que se apoyan en el servidor. Por supuesto, también hay desventajas. La más seria, que explicaremos en el capítulo 9, tiene que ver con la seguridad. Confiar en que el código descargado se implemente sólo en la interfaz anunciada mientras accede a su desprotegido disco duro, y no que envíe las partes más significativas a quién sabe dónde no siempre es una buena idea.

Modelos para migración de código

Aunque la migración de código sugiere que el código solamente se traslada a través de las máquinas, en realidad el término cubre un área mucho más completa. Tradicionalmente, en los sistemas distribuidos la comunicación va enfocada al intercambio de datos entre procesos. La migración de código desde el punto de vista de la difusión se basa en el movimiento de programas entre máquinas, con la intención de que dichos programas se ejecuten en el destino. En algunos casos, como en el proceso de migración, el estado de ejecución de un programa, señales pendientes, y otras partes del entorno también se deben trasladar.

Para tener un mejor entendimiento de los diferentes modelos existentes para la migración de código, utilizamos un marco de trabajo (framework) descrito en Fuggetta y colaboradores (1998). En este marco de trabajo, un proceso consta de tres segmentos. El *segmento de código* es la parte que contiene el conjunto de instrucciones constitutivas del programa en ejecución. El *segmento de recurso* contiene referencias a recursos externos necesarios para el proceso, tales como archivos, impresoras, dispositivos, otros procesos, entre otros factores. Finalmente, se usa un *segmento de ejecución* para almacenar el estado actual de la ejecución de un proceso, el cual consta de datos privados, la pila, y por supuesto, el contador de programa.

El mínimo necesario para una migración de código es proporcionar solamente **movilidad débil**. En este modelo, es posible transferir solamente el segmento de código, junto con quizás algunos datos de inicialización. Una característica clásica de la movilidad débil es que un programa transferido siempre comienza desde uno de los diversos puntos de inicio predefinidos. Por ejemplo, esto sucede con los applets de Java, los cuales siempre comienzan la ejecución desde el principio. El beneficio de este método es su simplicidad. La movilidad débil requiere solamente que una máquina de destino pueda ejecutar el código, y en esencia prepara al código para que sea portable. Regresaremos a estos temas cuando expliquemos la migración en sistemas heterogéneos.

Al contrario de la movilidad débil, en sistemas que soportan la **movilidad fuerte** el segmento de ejecución también se puede transferir. Esta característica clásica de la movilidad fuerte es que un proceso se puede detener, y posteriormente trasladarse hacia otra máquina, y luego continuar su ejecución a partir del punto en donde se quedó. Claramente, la movilidad fuerte es más general que la movilidad débil, pero también más difícil de implementar.

Sin importar si la movilidad es fuerte o débil, podemos hacer una distinción más profunda entre migración iniciada por el remitente y migración iniciada por el destinatario. En la migración **iniciada por el remitente**, por lo general, la migración empieza en la máquina donde reside el código o donde se ejecuta. A menudo, la migración iniciada por el remitente se hace cuando los programas se cargan al servidor de cómputo. Otro ejemplo, es el envío de un programa de búsqueda por internet hacia un servidor web de base de datos para ejecutar las consultas en dicho servidor. En la migración **iniciada por el destinatario**, la máquina de destino toma la iniciativa para realizar la migración de código. Los applets de Java son un ejemplo de este método.

La migración iniciada por el destinatario es más simple que la migración iniciada por el remitente. En muchos casos, la migración de código ocurre entre un cliente y un servidor, donde el cliente toma la iniciativa para realizar la migración. La carga segura del código hacia un servidor, como lo hace la migración iniciada por el remitente, con frecuencia requiere que el cliente se encuentre previamente registrado y autentificado en dicho servidor. En otras palabras, se requiere que el servidor conozca a todos sus clientes, la razón para esto es que el cliente presumiblemente querrá acceder a recursos de servidor tales como un disco. Proteger dichos recursos es esencial. Por el contrario, la descarga de código como en el caso iniciado por el destinatario, a menudo se realiza de manera anónima. Más aún, generalmente el servidor no está interesado en los recursos del cliente. En lugar de eso, la migración de código hacia el cliente solamente se hace para mejorar el rendimiento del lado del cliente. En ese punto, sólo es necesario proteger un número limitado de recursos, tales como la memoria y las conexiones de red. Regresaremos al código seguro más extensamente en el capítulo 9.

En el caso de la movilidad débil, también hay diferencia si el código migrado se ejecuta mediante el proceso de destino o es iniciado como un proceso separado. Por ejemplo, los applets de Java se descargan de manera sencilla mediante un navegador web y se ejecutan dentro del espacio de direcciones del navegador. El beneficio de este método es que no hay necesidad de comenzar un proceso por separado, ello evita la comunicación en la máquina de destino. La principal desventaja es que el proceso de destino requiere estar protegido contra ejecuciones de código maliciosas o inadvertidas. Una solución sencilla es dejar que el sistema operativo se haga cargo de eso mediante la creación de un proceso separado para ejecutar el código migrado. Observe que esta solución no

resuelve los problemas de acceso a recursos que mencionamos anteriormente. Aún tenemos que tratar con ellos.

En lugar de trasladar un proceso en ejecución, también conocido como migración de procesos, la movilidad fuerte puede ser soportada además por clonación remota. En contraste con la migración de procesos, clonar significa producir una copia exacta del proceso original, pero ahora ejecutado en una máquina diferente. El proceso de clonación se ejecuta en paralelo con el proceso original. En sistemas UNIX, la clonación remota tiene lugar al bifurcar un proceso hijo y dejar que el hijo continúe en una máquina diferente. El beneficio de la clonación es que el modelo se asemeja mucho al modelo que ya utilizan muchas aplicaciones. La única diferencia es que el proceso de clonación se ejecuta en una máquina diferente. En este sentido, la migración por clonación es una manera sencilla de mejorar la transparencia de distribución.

En la figura 3-18 aparecen las distintas alternativas para implementar la migración de código.

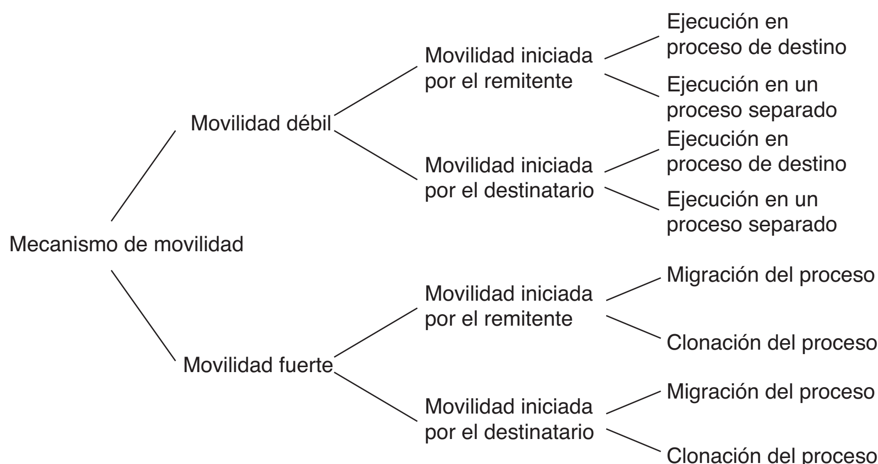


Figura 3-18. Alternativas para efectuar la migración de código.

3.5.2 Migración y recursos locales

Hasta aquí, solamente hemos tomado en cuenta la migración de código y los segmentos de ejecución. El segmento de recursos requiere atención especial. Con frecuencia, lo que vuelve tan difícil la migración de código es que el segmento de recursos no siempre se puede transferir de manera sencilla, junto con los otros segmentos, sin sufrir modificación alguna. Por ejemplo, supongamos que un proceso mantiene una referencia a un puerto TCP específico a través del cual se comunicaba con otros procesos (remotos). Tal referencia se mantiene dentro de su segmento de recurso. Cuando el proceso se traslada a otra ubicación, tendrá que entregarse el puerto y solicitarse uno nuevo en el destino. En otros casos, transferir una referencia no tiene que ser un problema. Por

ejemplo, una referencia a un archivo mediante una URL absoluta permanece válida independientemente de la máquina donde resida el proceso que mantiene la URL.

Para comprender las implicaciones que la migración de código tiene sobre el segmento de recursos, Fuggetta y colaboradores (1998) distinguen entre tres tipos de enlaces de proceso a recurso. El enlace más fuerte es cuando un proceso hace referencia a un recurso mediante su identificador. En ese caso, el proceso requiere precisamente del recurso al que se hace referencia, y nada más. Un ejemplo de dicho **enlace de identificador** es cuando un proceso utiliza una URL para hacer referencia al sitio de un web específico o cuando hace referencia a un servidor FTP mediante la dirección de internet de dicho servidor. En la misma línea de razonamiento, las referencias a los puntos finales de comunicación local ocasionan también un enlace mediante identificador.

Un modo más débil del enlace de proceso a recurso es cuando se requiere solamente el valor de un recurso. En ese caso, la ejecución del proceso no se verá afectada si otro recurso pudiera proporcionar ese mismo valor. Un ejemplo típico de **enlace por valor** es cuando el programa se apoya en bibliotecas estándar, tales como aquellas empleadas para la programación en C y Java. Dichas bibliotecas siempre deben estar disponibles de manera local, pero su ubicación exacta en el sistema de archivos local pudiera diferir entre sitios. No los archivos específicos, pero su contenido es importante para una ejecución apropiada del proceso.

Por último, la forma más débil de enlace es cuando un proceso indica que requiere solamente un recurso de un tipo específico. Este **enlace por tipo** se ejemplifica mediante referencias a dispositivos locales, tales como monitores, impresoras, y cosas por el estilo.

Cuando se migra código, con frecuencia requerimos modificar las referencias a los recursos, pero no podemos afectar el tipo de enlace de proceso a recurso. Si se debe modificar una referencia, y exactamente de qué manera hacerlo, depende de si es posible trasladar el recurso junto con el código hacia la máquina de destino. De manera más específica, necesitamos considerar los enlaces de recurso a máquina, y distinguir los siguientes casos. Los **recursos no adjuntos** se pueden trasladar fácilmente entre diversas máquinas, y por lo general los archivos (de datos) asociados sólo con el programa a migrar. Por el contrario, trasladar o copiar un **recurso adjunto** puede ser posible, pero sólo a un costo relativamente alto. Los ejemplos típicos de recursos adjuntos son las bases de datos locales y los sitios web completos. Aunque dichos recursos son, en teoría, independientes de su máquina actual, por lo general no es factible trasladarlos hacia otro ambiente. Por último, los **recursos fijos** están ligados de manera íntima a una máquina específica o a un ambiente y no se pueden trasladar. A menudo los recursos fijos son dispositivos locales. Otro ejemplo de un recurso fijo es un punto final local de comunicación.

Combinar tres tipos de enlaces de proceso a recurso, y tres tipos de enlaces de recurso a máquina, nos lleva a nueve combinaciones que necesitamos considerar al migrar el código. Podemos ver estas nueve combinaciones en la figura 3-19.

Consideremos primero las posibilidades de cuando un proceso está ligado a un recurso mediante un identificador. Cuando un recurso no está adjunto, por lo general es mejor trasladarlo junto con el código a migrar. Sin embargo, cuando el recurso es compartido por otros procesos, una alternativa es establecer una referencia global, esto es, una referencia que puede cruzar los límites de la máquina. Un ejemplo de dicha referencia es una URL. Cuando el recurso está adjunto o fijo, la mejor solución es crear también una referencia global.

Enlace recurso a máquina

		No adjunto	Adjunto	Fijo
Enlace proceso a recurso	Por identificador	MV (o GR)	GR (o MV)	GR
	Por valor	CP (o MV, GR)	GR (o CP)	GR
	Por tipo	RB (o MV, CP)	RB (o GR, CP)	RB (o GR)

GR Establece una referencia global de sistema
MV Traslada el recurso
CP Copia el valor del recurso
RV Reenlaza el proceso al recurso disponible de manera local

Figura 3-19. Acciones a emprender con respecto a las referencias a recursos locales al migrar el código hacia otra máquina.

Es importante darse cuenta de que establecer una referencia global puede ser más que solamente hacer uso de URL, y que el uso de dicha referencia en ocasiones es prohibitivamente costoso. Por ejemplo, considere un programa que genera imágenes de alta calidad para una estación de trabajo dedicada a multimedia. Fabricar imágenes de alta calidad en tiempo real es una tarea de cómputo intensivo, por cuya razón el programa pudiera ser trasladado hacia un servidor de cómputo de alto rendimiento. Establecer una referencia global para una estación de trabajo multimedia significa configurar una ruta de comunicación entre el servidor de cómputo y la estación de trabajo. Además, existe un proceso significativo involucrado tanto en el servidor como en la estación de trabajo para cumplir con los requerimientos del ancho de banda de las imágenes transferidas. El resultado neto pudiera ser que trasladar el programa del servidor a la computadora no sea buena idea, sólo porque el costo de la referencia global es muy alto.

Otro ejemplo de dónde establecer una referencia global no siempre es tan fácil al migrar un proceso que hace uso del punto final de la comunicación global. En ese caso, estamos tratando con un recurso fijo al cual el proceso se enlaza mediante un identificador. Existen básicamente dos soluciones. Una solución es permitir al proceso configurar una conexión hacia la máquina fuente después de migrar, e instalar un proceso separado en la máquina fuente que simplemente reenvíe todos los mensajes entrantes. La principal desventaja de este método es que cada vez que la máquina fuente falle, la comunicación con el proceso de migración pudiera interrumpirse. La solución alternativa es tener todos los procesos que se comunican con el proceso de migración, modificar *su* referencia global, y enviar mensajes al nuevo punto final de comunicación en la máquina destino.

La situación es diferente cuando tratamos con referencias por valor. Consideremos primero un recurso fijo. Por ejemplo, la combinación de un recurso fijo y una referencia por valor ocurre cuando un proceso asume que la memoria entre procesos puede compartirse. Establecer una referencia global, en este caso, podría significar que necesitamos implementar una forma distribuida de memoria compartida. En muchos casos, esto no resulta realmente en una solución viable o eficiente.

En general, los recursos adjuntos a los que se hace referencia por su valor son bibliotecas en tiempo de ejecución. Copias de dichos recursos están disponibles a menudo en la máquina destino, o de lo contrario se deben copiar antes de que ocurra la migración de código. Establecer una referencia global es una mejor alternativa cuando se van a copiar grandes cantidades de datos, como pudiera ser el caso con diccionarios y enciclopedias en sistemas de procesamiento de texto.

El caso más sencillo es al tratar con recursos no adjuntos. La mejor solución es copiar (o trasladar) el recurso al nuevo destino, a menos que se encuentre compartido por un número de proceso. En último caso, establecer una referencia global es la única alternativa.

El último caso trata con referencias por tipo. Sin tomar en cuenta la referencia recurso-máquina, la solución más evidente es la de volver a enlazar el proceso a un recurso local disponible del mismo tipo. Solamente cuando dicho recurso no esté disponible necesitaremos copiar o trasladar el original a un nuevo destino, o establecer una referencia global.

3.5.3 Migración y sistemas heterogéneos

Hasta aquí, asumimos de manera tácita que el código migrado se puede ejecutar fácilmente en la máquina destino. Esta suposición es correcta cuando lidiamos con sistemas homogéneos. Sin embargo, en general, los sistemas distribuidos están construidos sobre una colección de plataformas, cada plataforma tiene su propio sistema operativo y arquitectura de máquina. La migración en dichos sistemas requiere que cada plataforma se encuentre soportada, esto es, que el segmento de código se pueda ejecutar en cada plataforma. Además, necesitamos asegurarnos de que el segmento de ejecución se pueda representar apropiadamente en cada plataforma.

Los problemas que acarrea la heterogeneidad son los mismos, en muchos aspectos, que aquellos referentes a la portabilidad. Sin ser sorprendente, las soluciones también resultan muy similares. Por ejemplo, al final de la década de 1970, una solución sencilla para mitigar muchos de los problemas de portar Pascal hacia diferentes máquinas era generar código máquina independiente de la máquina para una máquina virtual abstracta (Barron, 1981). Por supuesto, esa máquina necesitaría implementarse en muchas plataformas, pero entonces permitiría la ejecución de los programas escritos en Pascal en cualquier parte. Aunque esta sencilla idea fue utilizada ampliamente por algunos años, nunca se consideró como solución general para los problemas de portabilidad de otros lenguajes, como C.

Unos 25 años después, la migración de código en sistemas heterogéneos es enfrentada por lenguajes mediante scripts altamente portables como Java. En esencia, estas soluciones adoptan el mismo método aplicado para portar Pascal. Todas tienen en común que se apoyan en una máquina virtual (procesamiento) que interpreta el código fuente de manera directa (como en el caso de los leguajes que implementan scripts), o de lo contrario interpreta código intermedio mediante el compilador (como en Java). Estar en el lugar indicado en el momento indicado también es importante para los desarrolladores de lenguajes.

Los desarrollos recientes han comenzado a debilitar la dependencia de los lenguajes de programación. En especial, las soluciones se propusieron no solamente para procesos de migración, sino

también para migrar ambientes de programación completos. La idea básica es dividir en compartimentos (fragmentos) todo el ambiente y los procesos para su migración como si se tratase de parte de su ambiente de cómputo.

Si la división en compartimentos se realiza de manera adecuada, será posible desacoplar una parte del sistema subyacente y realmente migrarla hacia otra máquina. De esta manera, la migración proporcionará en efecto una forma de movilidad fuerte para los procesos, dado que es posible trasladarla hacia cualquier punto durante su ejecución, y continuar en donde se quedó cuando se complete la migración. Más aún, podemos resolver muchos de los asuntos intrincados relacionados con la migración de procesos mientras éstos cuenten con enlaces hacia recursos locales. Los recursos locales a menudo son parte del ambiente que se está migrando.

Existen distintas razones para esperar a la migración de ambientes completos, pero quizá la más importante es que permite la continuación de la operación mientras alguna máquina necesita detener su ejecución. Por ejemplo, dentro de un servidor de cluster, los administradores de sistemas pudieran decidir apagar o reemplazar una máquina, pero no tendrán que detener todos sus procesos en ejecución. En su lugar, pudieran congelar un ambiente de manera temporal, trasladarlo a otra máquina (en donde esté junto a otro ambiente existente), y simplemente descongelarlo de nuevo. De manera clara, ésta es una forma muy poderosa de administrar grandes ambientes de cómputo en ejecución y sus procesos.

Consideremos ahora un ejemplo específico para migrar máquinas virtuales, como explican Clark y colaboradores (2005). En este caso, los autores se concentran en la migración en tiempo real de un sistema operativo virtual, típicamente algo que pudiera ser apropiado para un servidor de cluster con acoplamiento estrecho es mejorar su rendimiento a través de una red de área local individual. Bajo estas circunstancias, la migración involucra dos problemas importantes: migrar la imagen completa de la memoria y migrar los enlaces a recursos locales.

Tal como en el primer problema, existen, en principio, tres maneras de manipular la migración (que pueden ser combinadas):

1. Empujar las páginas de memoria hacia la nueva máquina y reenviar las que se modificaron posteriormente durante el proceso de migración.

2. Detener la máquina virtual actual; migrar la memoria, y comenzar una nueva máquina virtual.

3. Dejar que la nueva máquina virtual atraiga las nuevas páginas cuando sea necesario, esto es, permitir que los procesos comiencen dentro de una nueva máquina virtual de inmediato y copiar las páginas de memoria según la demanda.

La segunda alternativa pudiera generar un tiempo de espera inaceptable si la máquina virtual ejecuta un servicio en vivo, esto es, que ofrece un servicio continuo. Por otro lado, el puro método según la demanda está representado por la tercera alternativa y pudiera prolongar extensamente el periodo de migración, pero también pudiera provocar un pobre rendimiento debido a que toma mucho tiempo antes de que el conjunto de procesos de migración se traslade hacia una nueva máquina.

Como alternativa, Clark y colaboradores (2005) proponen el uso de un método previo de copiado, el cual combina la primera alternativa, junto con una breve fase de parar y copiar representada por la segunda modalidad. Tal como surge de lo anterior, esta combinación puede provocar servicios con tiempos de espera de hasta 200 ms o menos.

Con respecto a los recursos locales, las cuestiones se simplifican al tratar solamente con un cluster de servidores. Primero, debido a que existe una sola red, lo único que necesita hacerse es anunciar el nuevo enlace de la dirección "MAC a la red", de modo que los clientes puedan hacer contacto con los procesos de migración en la nueva interfaz de red correcta. Finalmente, si podemos asumir que el almacenamiento se proporciona como una capa por separado (según indica la figura 3-12), entonces la migración de los enlaces hacia los archivos es similarmente sencilla.

El efecto general es que, en lugar de migrar los procesos, sabemos que podemos trasladar entre máquinas al sistema operativo completo.

3.6 RESUMEN

Los procesos juegan un papel fundamental en los sistemas distribuidos ya que forman la base para la comunicación entre máquinas diferentes. Un tema importante es cómo se organizan y, en especial, si soportan o no varios hilos de control. En los sistemas distribuidos, los hilos son particularmente útiles para continuar usando la CPU al llevarse a cabo el bloqueo de la operación de E/S. De esta manera, es posible la construcción de servidores altamente eficientes que ejecutan diferentes hilos en paralelo, de los cuales muchos se pueden bloquear para esperar hasta que se complete una operación en el disco de E/S o se desbloquee la comunicación en red.

La organización de una aplicación distribuida en términos de clientes y servidores ha probado ser útil. Los procesos del cliente por lo general implementan interfaces de usuario, las cuales comprenden desde sencillos despliegues hasta interfaces que pueden manipular documentos compuestos. Además, es más probable que el software cliente logre obtener la transparencia de distribución al ocultar los detalles de comunicación con los servidores, cuando dichos servidores están localizados, y sean o no servidores replicados. Asimismo, el software cliente es parcialmente responsable de ocultar las fallas y de recuperarse de las fallas.

Por lo general, los servidores son más intrincados que los clientes, sin embargo solamente son sujetos de un número relativamente pequeño de problemas de diseño. Por ejemplo, los servidores pueden ser iterativos o concurrentes, implementar uno o varios servicios, y pueden ser con estado o sin estado. Otros problemas de diseño tratan con los servicios de direccionamiento y con los mecanismos para interrumpir un servidor después de haber solicitado un servicio, y que posiblemente ya se procesó.

Debemos poner especial atención al organizar los servidores dentro de un cluster. Un objetivo común es ocultar las características internas de un cluster del mundo exterior. Esto significa que la organización de un cluster debe estar protegida de las aplicaciones. Hasta este punto, la mayoría de los clústeres utiliza un solo punto de entrada que puede manipular los mensajes hacia los servidores del cluster. Un problema desafiante es el de reemplazar este único punto de entrada para que sea una solución distribuida completa.

Un tema importante en los sistemas distribuidos es la migración de código entre máquinas diferentes. Dos razones principales para soportar la migración de código son la mejora del rendimiento y la flexibilidad. En ocasiones, cuando la comunicación es cara, podemos reducirla enviando los cálculos desde el servidor hacia el cliente, y dejar que el cliente haga el mayor procesamiento local posible. La flexibilidad se incrementa si un cliente puede descargar de manera dinámica el software necesario para comunicarse con el servidor. El software de descarga puede direccionarse específicamente hacia dicho servidor, sin tener que forzar al cliente a instalarlo previamente.

La migración de código genera tres problemas relacionados con el uso de recursos locales, por lo cual se requiere que los recursos se migren también, que se establezcan nuevos enlaces hacia los recursos locales en la máquina destino, o que se utilicen referencias para toda la red. Otro problema es que la migración de código requiere que tomemos en cuenta la heterogeneidad. La práctica común indica que la mejor solución para manipular la heterogeneidad es crear máquinas virtuales. Por ejemplo, esto pudiera tomar tanto la forma de máquinas virtuales de proceso como en el caso de Java, o a través de máquinas virtuales de monitoreo que permiten efectivamente la migración de una colección de procesos junto con su sistema operativo subyacente.

PROBLEMAS

1. En este problema usted debe comparar un archivo de lectura mediante un servidor de archivos de un solo hilo y un servidor multihilos. Toma 15 milisegundos hacer que una petición se ejecute, despache, y haga el resto del proceso necesario, asumiendo que los datos pertinentes se encuentran en un caché en la memoria principal. Si una operación de disco es necesaria, como es el caso de un tercio del tiempo, se requieren 75 milisegundos adicionales, durante los cuales el hilo permanece dormido. ¿Cuántas peticiones por segundo puede manipular el servidor si contiene un solo hilo? ¿Y si es multihilos?

2. ¿Tendría sentido limitar el número de hilos en el proceso del servidor?

3. En el texto, describimos un servidor de archivos multihilos, y mostramos por qué es mejor que un servidor de un solo hilo y un servidor de máquina de estado finito. ¿Existe alguna circunstancia en la cual un servidor de un solo hilo pudiera ser mejor? Dé un ejemplo.

4. Asociar de manera estática solamente un hilo con un proceso de peso ligero no es una buena idea. ¿Por qué?

5. Tener sólo un proceso de peso ligero por proceso tampoco es buena idea. ¿Por qué?

6. Describa un esquema simple en el cual existan tanto procesos de peso ligero como hilos ejecutables.

7. X designa una terminal de usuario como host del servidor, mientras que a la aplicación se le conoce como el cliente. ¿Tiene sentido lo anterior?

8. El protocolo X sufre de problemas de escalabilidad. ¿Cómo podemos enfrentar estos problemas?

9. Los *proxies* pueden soportar transparencia de replicación mediante invocación de cada réplica, como explicamos en el texto. ¿Podrá (del lado del servidor) una aplicación estar sujeta a las réplicas de las llamadas?

10. La construcción de un servidor concurrente mediante la división de un proceso tiene algunas ventajas y desventajas comparada con los servidores multihilos. Mencione algunas de tales ventajas y desventajas.

11. Diseñe un servidor multihilos que soporte múltiples protocolos mediante el uso de canales de comunicación (sockets) como si fuera una interfaz a nivel de transporte para el sistema operativo subyacente.

12. ¿Cómo podemos prevenir que una aplicación evite un administrador de ventanas, y de esta manera pueda arruinar por completo una pantalla?

13. Sea un servidor que da mantenimiento a un cliente mediante una conexión TCP/IP, ¿será un caso de un **servidor con estado** o **servidor sin estado**?

14. Imagine un servidor web que da mantenimiento a una tabla en la cual están mapeadas las direcciones IP de los clientes que accedieron a las páginas más recientemente consultadas. Cuando un cliente se conecta a un servidor, el servidor busca al cliente en su tabla, y si lo encuentra, devuelve la página registrada. ¿Es este servidor **con estado** o **servidor sin estado**?

15. La fuerte movilidad en sistemas UNIX debiera ser soportada al permitir a un proceso que migre un hijo en una máquina remota. Explique cómo funcionaría esto.

16. En la figura 3-18 sugerimos que la movilidad fuerte no se puede combinar con la ejecución de código migrado para un proceso. Proporcione un ejemplo.

17. Considere un proceso *P* que requiere acceso al archivo *F* disponible de manera local en la máquina donde se ejecuta *P*. Cuando trasladamos *P* a otra máquina, aún requerimos acceso a *F*. Si corregimos el vínculo archivo-máquina, ¿cómo podemos implementar la referencia a *F*?

18. Describa con detalle la manera en que fluyen los paquetes TCP en el caso del TCP handoff, junto con la información que respecta a las direcciones fuente y destino en los distintos encabezados.

4

COMUNICACIÓN

La comunicación entre procesos se encuentra en el núcleo de todos los sistemas distribuidos. No tiene sentido estudiar los sistemas distribuidos, sin examinar cuidadosamente las formas en que los procesos desarrollados en diferentes máquinas pueden intercambiar información. En los sistemas distribuidos, la comunicación siempre se basa en el paso de mensajes de bajo nivel, tal como lo ofrece la red subyacente. Expresar la comunicación a través del paso de mensajes, es más difícil que utilizar primitivas basadas en memoria compartida, como se hace en plataformas no distribuidas. Los sistemas distribuidos modernos, con frecuencia consisten en miles o incluso millones de procesos esparcidos en una red con comunicación poco confiable como internet. A menos que las primitivas de comunicación para redes de computadoras sean remplazadas por algo más, el desarrollo de aplicaciones de gran escala es extremadamente difícil.

En este capítulo, iniciamos nuestro estudio explicando las reglas a que deben apegarse los procesos de comunicación, conocidos como protocolos, y nos concentraremos en estructurarlos en forma de capas. Después veremos tres modelos ampliamente utilizados para efectuar la comunicación: la llamada a un procedimiento remoto (RPC), middleware orientado a mensajes (MOM), y flujo de datos. También explicaremos el problema general de enviar datos a diversos destinatarios, llamado multitransmisión (multicasting).

Nuestro primer modelo para comunicación en sistemas distribuidos, es la llamada a un procedimiento remoto (RPC, por sus siglas en inglés). Una RPC intenta ocultar lo intrincado del paso de mensajes, y es ideal para aplicaciones cliente-servidor.

En muchas aplicaciones distribuidas, la comunicación no sigue el estricto patrón de interacción cliente-servidor. En esos casos, resulta evidente que pensar en términos de mensajes es más adecuado.

Sin embargo, las herramientas de comunicación de bajo nivel de las redes de computadoras no son, en muchos sentidos, adecuadas debido a su carencia de transparencia de distribución. Una alternativa, es utilizar un modelo de cola de mensajes de alto nivel, en el que la comunicación se realiza de forma muy parecida a los sistemas de correo electrónico. El middleware orientado a mensajes (MOM, por sus siglas en inglés) es un tema lo suficientemente importante como para dedicarle una sección completa.

Con la llegada de sistemas distribuidos multimedia, se hizo evidente que muchos sistemas carecían de soporte para comunicación de medios continuos, tal como audio y video. Lo que se necesita es la idea de un medio que pueda soportar el flujo continuo de mensajes, sujeto a diversas restricciones de tiempo. En otra sección explicaremos los flujos.

Por último, debido a que nuestra comprensión para configurar herramientas de multitransmisión ha mejorado, han surgido novedosas y elegantes soluciones para la diseminación de datos. Aplicaremos nuestra atención por separado a esta materia en la última sección de este capítulo.

4.1 FUNDAMENTOS

Antes de iniciar nuestra explicación sobre la comunicación en sistemas distribuidos, primero recapitularemos sobre algunas cuestiones fundamentales relacionadas con la comunicación. En la siguiente sección, analizaremos brevemente los protocolos de comunicación de redes, ya que constituyen la base para cualquier sistema distribuido. Después, aplicamos un enfoque diferente clasificando los diferentes tipos de comunicación que tienen lugar en los sistemas distribuidos.

4.1.1 Protocolos en capas

Debido a la ausencia de memoria compartida en los sistemas distribuidos, toda la comunicación se basa en el envío y la recepción (de bajo nivel) de mensajes. Cuando el proceso *A* debe comunicarse con el proceso *B*, primero elabora un mensaje en su propio espacio de dirección. Después ejecuta una llamada de sistema y ocasiona que el sistema operativo envíe el mensaje sobre la red hacia *B*. Aunque esta idea básica parece bastante sencilla, para evitar el caos, *A* y *B* deben acordar el significado de los bits por enviar. Si *A* envía una nueva y brillante novela escrita en francés, y codificada con el código de caracteres EBCDIC de IBM, y *B* espera el inventario de un supermercado escrito en inglés, y codificado en ASCII, la comunicación será menos que óptima.

Se necesitan muchos acuerdos diferentes. ¿Cuántos volts deben utilizarse para representar un bit 0, y cuántos para representar un bit 1? ¿Cómo sabe el destinatario cuál es el último bit del mensaje? ¿Cómo puede detectar si un mensaje se ha dañado o perdido, y qué debe hacer si lo descubre? ¿Qué tan largos son los números, cadenas, y otros elementos de datos, y cómo se representan? En resumen, los acuerdos son necesarios en diversos niveles que van desde detalles de bajo nivel sobre la transmisión de bits, hasta los detalles de alto nivel sobre cómo va a expresarse la información.

Para facilitar el manejo de varios niveles y cuestiones involucradas en la comunicación, la International Standards Organization (ISO) desarrolló un modelo de referencia que identifica claramente los diversos niveles involucrados, les proporciona nombres estándar, y señala qué nivel debe hacer cuál trabajo. Este modelo se conoce como el **Modelo de Referencia de Interconexión de Sistemas Abiertos** (Day y Zimmerman, 1983), normalmente abreviado como **ISO OSI** o bien, en ocasiones, sólo como el **modelo OSI**. Es necesario resaltar que los protocolos desarrollados como parte del modelo OSI nunca se utilizaron ampliamente, y que hoy en día están prácticamente muertos. Sin embargo, el modelo subyacente ha mostrado ser muy útil para comprender las redes de computadoras. Aunque no intentamos dar una descripción completa de este modelo y de todas sus implicaciones, una breve introducción será útil. Para conocer más detalles, consulte Tanenbaum (2003).

El modelo OSI está diseñado para permitir que los sistemas abiertos se comuniquen. Un sistema abierto, es aquel que está preparado para comunicarse con cualquier otro sistema abierto mediante reglas estándar que regulen formato, contenido, y significado de los mensajes enviados y recibidos. Estas reglas están formalizadas en lo que conocemos como **protocolos**. Para permitir a un grupo de computadoras comunicarse a través de una red, deben acordarse los protocolos a utilizar. Se hace una diferencia entre dos tipos generales de protocolos. Con los protocolos **orientados a conexión**, antes de intercambiar datos, el remitente y el destinatario primero establecen explícitamente una conexión, y posiblemente negocian el protocolo que utilizarán. El teléfono es un sistema de comunicación orientado a conexión. Con los protocolos orientados a no **conexión**, no se necesita una configuración por adelantado. El remitente sólo transmite el primer mensaje cuando está listo. Dejar una carta en el buzón es un ejemplo de comunicación orientada a no conexión. Con las computadoras, tanto la comunicación orientada a conexión como la que no son orientadas a conexión son comunes.

En el modelo OSI, la comunicación se divide en siete niveles o capas, como ilustra la figura 4-1. Cada capa maneja un aspecto específico de la comunicación. En este sentido, el problema puede dividirse en partes manejables, cada una de las cuales puede resolverse de manera independiente de las otras. Cada capa proporciona una interfaz hacia la siguiente capa superior. La interfaz consiste en un conjunto de operaciones que definen el servicio que la capa podrá ofrecer a sus usuarios.

Cuando el proceso *A* de la máquina 1 debe comunicarse con el proceso *B* de la máquina 2, elabora un mensaje y lo pasa hacia la capa de aplicación localizada en su máquina. Por ejemplo, esta capa puede ser un procedimiento de biblioteca, aunque también podría implementarse de otra manera (digamos, dentro del sistema operativo, en un procesador externo de red, etc.). El software de la capa de aplicación en turno agrega entonces un **encabezado** al frente del mensaje, y pasa el mensaje resultante a través de la interfaz de la capa 6/7 hacia la capa de presentación. La capa de presentación en turno agrega su propio encabezado y pasa el resultado hacia abajo, hacia la capa de sesión, y así sucesivamente. Algunas capas no sólo agregan un encabezado al frente, sino que también agregan una parte al final. Cuando éste toca el fondo, la capa física transmite entonces el mensaje (que ahora podría lucir como el de la figura 4-2) colocándolo en un medio de transmisión física.

Cuando el mensaje llega a la máquina 2, pasa hacia arriba, quitando cada capa y examinando su propio encabezado. Por último, el mensaje llega al destinatario, el proceso *B*, el cual puede res-

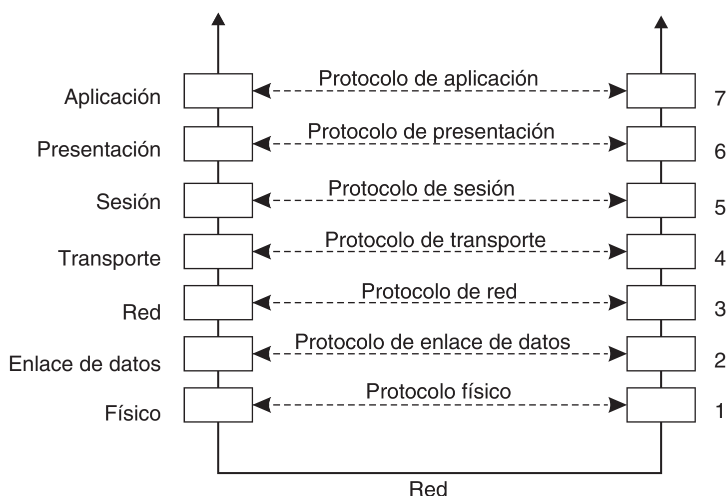


Figura 4-1. Capas, interfaces y protocolos del modelo OSI.

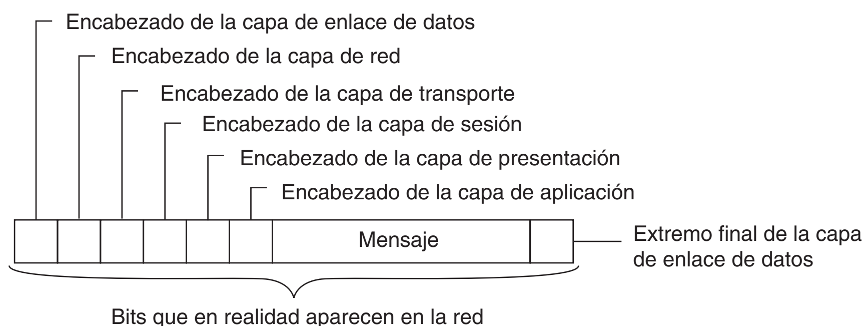


Figura 4-2. Mensaje típico de acuerdo a como aparece en la red.

ponder utilizando la ruta inversa. La información del encabezado de la capa n se utiliza para el protocolo de la capa n.

Como ejemplo del por qué los protocolos son importantes, consideremos la comunicación entre dos empresas, Zippy Airlines y su proveedor Mushy Meals, Inc. En Zippy, cada mes el director de servicios a los pasajeros solicita a su secretaria contactar a la secretaria del gerente de ventas de Mushy para ordenar 100 000 cajas de pollo empanizado. Tradicionalmente, las órdenes se enviaban mediante la oficina postal, sin embargo, debido a que el servicio postal se fue deteriorando, en algún momento ambas secretarias decidieron abandonarlo y comunicarse por correo electrónico. Ellas pudieron hacerlo sin molestar a sus jefes, ya que sus protocolos tratan con la transmisión física de las órdenes, no con sus contenidos.

De manera similar, el director de servicio a los pasajeros puede decidir abandonar el pollo empanizado, e ir por algo nuevo y especial para Mushy, costilla de cabra de primera, sin que esa decisión afecte a las secretarias. La cuestión es observar que aquí tenemos dos capas, los jefes y las secretarias. Cada capa tiene su propio protocolo (sujetos de análisis y tecnología) que puede modificarse de modo independiente del otro. Es precisamente esta independencia lo que hace atractivos a los protocolos. Cada protocolo puede cambiarse cuando la tecnología mejore, sin afectar a otros protocolos.

En el modelo OSI no hay dos capas, sino siete, como vemos en la figura 4-1. La colección de protocolos utilizados en un sistema particular se conoce como **suite de protocolos** o **pila de protocolos**. Es importante saber distinguir un *modelo de referencia* de sus *protocolos*. Como ya mencionamos, los protocolos OSI nunca fueron populares. Por contraste, los protocolos desarrollados para internet, como TCP e IP, son los más utilizados. En las siguientes secciones, analizaremos brevemente cada una de las capas OSI, a partir del fondo. Sin embargo, en lugar de proporcionar ejemplos de protocolos OSI, en donde sea adecuado señalaremos algunos protocolos de internet utilizados en cada capa.

Protocolos de bajo nivel

Iniciemos con la explicación de las tres capas más bajas de la suite de protocolos de OSI. Juntas, estas capas implementan las funciones básicas que forman una red de computadoras.

La capa física se ocupa de transmitir ceros y unos. Cuántos volts se utilizan para un 0 y cuántos para un 1, cuántos bits por segundo se pueden enviar, y si la transmisión puede ocurrir simultáneamente en ambas direcciones, son cuestiones clave de la capa física. Además, el tamaño y la forma del conector de red (plug), así como el número de pins y el significado de cada pin, son importantes aquí.

El protocolo de la capa física maneja la estandarización eléctrica y mecánica y la señalización de interfaces, de tal modo que cuando una máquina envía un bit 0, éste en realidad se recibe como un bit 0, y no como un bit 1. Se han desarrollado muchos estándares para la capa física (para diferentes medios), por ejemplo, el estándar RS-232-C para líneas de comunicación en serie.

La capa física sólo envía bits. Mientras no ocurran errores, todo está bien. Sin embargo, las redes de comunicación reales son propensas a errores, por lo que se necesita algún mecanismo para detectarlos y corregirlos. Este mecanismo es la tarea principal de la capa de enlace de datos; lo que hace es agrupar los bits en unidades, algunas veces llamadas **tramas**, y ve que cada trama se reciba correctamente.

La capa de enlace de datos hace su trabajo, colocando un patrón especial de bits al inicio y al final de cada trama para marcarlas, así como calculando una **suma de verificación** en la que suma de cierta manera todos los bytes de la trama. La capa de enlace de datos agrega a la trama la suma de verificación. Cuando la trama llega, el destinatario vuelve a calcular la suma de verificación de los datos, y compara el resultado con la suma de verificación que sigue a la trama. Si ambas sumas coinciden, la trama se considera correcta y es aceptada; si no coinciden, el destinatario solicita al remitente que la retransmita. A las tramas se les asignan números en secuencia (en el encabezado), para que todas puedan informar cuál es cuál.

En una LAN, por lo general no hay necesidad de que el remitente localice al destinatario. Éste simplemente coloca el mensaje en la red y el destinatario lo recibe. Sin embargo, una red de área amplia consiste en una gran cantidad de máquinas, cada una con cierto número de líneas hacia otras máquinas, parecida a un mapa a gran escala con las principales ciudades y caminos que las conectan. Para que un mensaje llegue del remitente al destinatario, es probable que deba realizar una serie de saltos, y elegir en cada salto una línea de salida. La cuestión de cómo elegir la mejor ruta se conoce como **enrutamiento**, y es básicamente la tarea principal de la capa de red.

El problema es complicado, debido a que la ruta más corta no siempre resulta ser la mejor. Lo que realmente importa es el monto del retraso en una ruta dada, lo cual, a su vez, se relaciona con el tráfico y el número de mensajes ubicados en la cola para transmitir por las diversas líneas. El retraso puede cambiar entonces con el transcurso del tiempo. Algunos algoritmos de enrutamiento intentan adaptarse a cargas cambiantes, mientras que otros se conforman con tomar decisiones basadas en promedios de largo plazo.

En la actualidad, el protocolo de red más ampliamente utilizado es el **IP** (**Protocolo de Internet**), el cual forma parte de la suite de protocolos de internet. Un **paquete** IP (el término técnico para un mensaje en la capa de red) puede enviarse sin configuración alguna. Cada paquete IP se pone en ruta hacia su destino, independientemente de todos los demás paquetes. Ninguna ruta interna se selecciona ni recuerda.

Protocolos de transporte

La capa de transporte forma la última parte de lo que podríamos llamar una pila básica de protocolos de red en el sentido de que implementa todos aquellos servicios no proporcionados en la interfaz de la capa de red, pero que son razonablemente necesarios para construir aplicaciones de red. En otras palabras, la capa de transporte transforma la red subyacente en algo que un desarrollador de aplicaciones puede utilizar.

Los paquetes pueden perderse en el camino del remitente hacia el destinatario. Aunque algunas aplicaciones pueden manejar su propia recuperación de errores, otras prefieren tener una conexión confiable. El trabajo de la capa de transporte es proporcionar este servicio. La idea es que la capa de aplicación debe poder entregar un mensaje a la capa de transporte con la expectativa de que será entregado sin pérdidas.

Una vez que recibe un mensaje desde la capa de aplicación, la capa de transporte lo divide en piezas lo suficientemente pequeñas como para transmitirlas, asigna a cada pieza un número secuencial, y después las envía todas. El análisis efectuado en el encabezado de la capa de transporte tiene que ver con qué paquetes se han enviado, cuáles se han recibido, para cuántos más tiene espacio el destinatario, cuáles deben retransmitirse, y otros aspectos similares.

Las conexiones de transporte confiables (que por definición están orientadas a conexión) pueden construirse hasta arriba de los servicios de red orientados a conexión o de red orientada a no conexión. En el primer caso, todos los paquetes llegarán en la secuencia correcta (si llegan todos), pero en el último caso, es posible que un paquete tome una ruta diferente y llegue primero que un paquete enviado antes. Depende del software de la capa de transporte el colocar todo en orden nuevamente, para mantener la ilusión de que una conexión de transporte es como un gran

tubo —usted coloca mensajes en él y salen sin daños y en el mismo orden en que se fueron. Proporcionar este comportamiento de comunicación fin a fin es un aspecto importante de la capa de transporte.

El protocolo de transporte de internet es conocido como **TCP** (**Protocolo para el Control de Transmisiones**), y se describe con detalle en Comer (2006). La combinación TCP-IP se utiliza ahora como un estándar predeterminado para comunicación en red. La suite de protocolos de internet también da soporte a un protocolo de transporte orientado a no conexión, llamado **UDP** (Protocolo Universal Datagram), el cual esencialmente es el IP con algunas adiciones menores. Los programas de usuario que no necesitan un protocolo orientado a conexión normalmente utilizan un UDP.

En general, se proponen otros protocolos de transporte; por ejemplo, para soportar la transferencia de datos en tiempo real, se definió el **Protocolo de Transporte en Tiempo Real** (**RTP**, por sus siglas en inglés). El RTP es un protocolo de red en el sentido de que especifica formatos de paquetes para datos en tiempo real sin proporcionar un mecanismo que garantice la entrega de los datos. Además, especifica un protocolo para monitorear y controlar la transferencia de datos de paquetes RTP (Schulzrinne y cols., 2003).

Protocolos de más alto nivel

Por encima de la capa de transporte, OSI diferenció tres capas adicionales. En la práctica, únicamente la capa de aplicación se utiliza siempre. De hecho, en la suite de protocolos de internet, lo que se encuentra por encima de la capa de transporte se agrupa todo junto. De cara a los sistemas middleware, en esta sección veremos que ni el enfoque OSI ni el de internet son realmente adecuados.

La capa de sesión es básicamente una versión mejorada de la capa de transporte. La capa de sesión proporciona control de diálogo para dar seguimiento a la parte que está comunicando en el momento, también proporciona herramientas de sincronización. Éstas son útiles para que los usuarios inserten puntos de verificación en transferencias largas, de tal modo que cuando sucede un desperfecto, sólo es necesario regresar al último punto de verificación, y no hasta el comienzo. En la práctica, pocas aplicaciones se interesan en la capa de sesión, y raramente es soportada. Incluso no está presente en la suite de protocolos de internet. Sin embargo, en el contexto de desarrollar soluciones middleware, el concepto de una sesión y sus protocolos relacionados se han vuelto muy importantes, principalmente cuando definen protocolos de comunicación de más alto nivel.

A diferencia de capas más bajas, que tienen que ver con la obtención de bits del remitente al destinatario de manera confiable y eficiente, la capa de presentación tiene que ver con el significado de los bits. La mayoría de los mensajes no consiste en cadenas de bits aleatorios, sino en información más estructurada tal como el nombre de una persona, direcciones, cantidades de dinero, etc. En la capa de presentación es posible definir registros que contengan campos como éstos, y después hacer que el remitente notifique al destinatario que un mensaje contiene un registro particular en cierto formato. Esto hace que para las máquinas con representaciones internas diferentes resulte más sencillo comunicarse una con otra.

La capa de aplicación de OSI, originalmente se proyectó para mantener una colección estándar de aplicaciones de red, como las de correo electrónico, transferencia de archivos, y emulación de terminales. Hasta el momento, se ha vuelto el contenedor de todas las aplicaciones y protocolos que, de una u otra manera, no encajan en una de las capas subyacentes. Desde la perspectiva del modelo de referencia OSI, todos los sistemas distribuidos son virtualmente sólo aplicaciones.

Lo que le falta a este modelo es una clara diferencia entre aplicaciones, protocolos específicos de aplicaciones, y protocolos de propósito general. Por ejemplo, el **Protocolo de Transferencia de Archivos** de internet (**FTP**) (Postel y Reynolds, 1985; y Horowitz y Lunt, 1997) define un protocolo para transferencia de archivos entre una máquina cliente y una servidor. El protocolo no debe confundirse con el programa *ftp*, que es una aplicación de usuario final para transferencia de archivos, y que también resulta ser (no totalmente por coincidencia) para implementar el FTP de internet.

Otro ejemplo de protocolo típico de una aplicación específica, es el **Protocolo de Transferencia de Hipertexto** (**HTTP**) (Fielding y cols., 1999), el cual está diseñado para administrar y manejar remotamente la transferencia de páginas web. El protocolo se implementa mediante aplicaciones como navegadores web y servidores web. Sin embargo, ahora también los sistemas que no están intrínsecamente unidos a la web utilizan HTTP. Por ejemplo, el mecanismo de invocación de objetos de Java utiliza HTTP para solicitar la invocación de objetos remotos protegidos por un firewall (Sun Microsystems, 2004b).

También, existen muchos otros protocolos de propósito general que son útiles para muchas aplicaciones, pero no pueden calificarse como protocolos de transporte. En muchos casos, tales protocolos caen en la categoría de protocolos middleware, los cuales explicaremos a continuación.

Protocolos middleware

El middleware es una aplicación que lógicamente reside (la mayor parte del tiempo) en la capa de aplicación, pero que contiene muchos protocolos de propósito general que garantizan sus propias capas, independientemente de otras aplicaciones más específicas. Podemos establecer cierta diferencia entre los protocolos de comunicación de alto nivel y los protocolos implementados para establecer diversos servicios middleware.

Existen numerosos protocolos para soportar toda una variedad de servicios middleware. Por ejemplo, como explicaremos en el capítulo 9, hay varias formas de establecer la autenticación, es decir, proporcionar evidencia de una identidad declarada. Los protocolos de autenticación no están muy relacionados con alguna aplicación específica, pero pueden estar integrados en un sistema middleware como un servicio general. De igual manera, los protocolos de autorización mediante los cuales se garantiza el acceso a usuarios y procesos autenticados, a los recursos para los que tienen autorización, tienden a ser de naturaleza general e independientes de aplicaciones.

Como otro ejemplo, en el capítulo 8 consideraremos algunos protocolos de confirmación distribuidos. Los protocolos de confirmación establecen que, en un grupo de procesos, o todos los procesos realizan una operación en particular o la operación no se realiza en absoluto. Este fenómeno, también se conoce como **atomicidad**, y se aplica ampliamente en transacciones. Como veremos,

además de las transacciones, otras aplicaciones, como las que toleran fallas, pueden aprovechar los protocolos de confirmación distribuidos.

Como un último ejemplo, consideremos un protocolo distribuido de aseguramiento mediante el cual un recurso puede ser protegido contra accesos simultáneos por una colección de procesos distribuidos a través de diversas máquinas. En el capítulo 6 trataremos varios de estos protocolos. De nuevo, éste es un ejemplo de un protocolo que puede utilizarse para implementar un servicio middleware general, pero que al mismo tiempo es muy independiente de cualquier aplicación específica.

Los protocolos de comunicación middleware soportan servicios de comunicación de alto nivel. Por ejemplo, en las dos siguientes secciones explicaremos protocolos que permiten que un proceso llame a un procedimiento o invoque a un objeto ubicado en una máquina remota de manera muy transparente. De igual forma, existen servicios de comunicación de alto nivel para establecer y sincronizar flujos para transferencia de datos en tiempo real, tales como los necesarios para aplicaciones multimedia. Como un último ejemplo, algunos sistemas middleware ofrecen servicios confiables de multidifusión que escalan a miles de destinatarios esparcidos en una red de área amplia.

Algunos de los protocolos de comunicación middleware podrían pertenecer a la capa de transporte, pero es posible que existan razones específicas para mantenerlos a un nivel más alto. Por ejemplo, los confiables servicios de multidifusión que garantizan la escalabilidad pueden implementarse sólo cuando se consideran los requerimientos de la aplicación. En consecuencia, un sistema middleware puede ofrecer diferentes protocolos (ajustables), que a su vez se implementan utilizando diferentes protocolos de transporte, pero ofreciendo una sola interfaz.

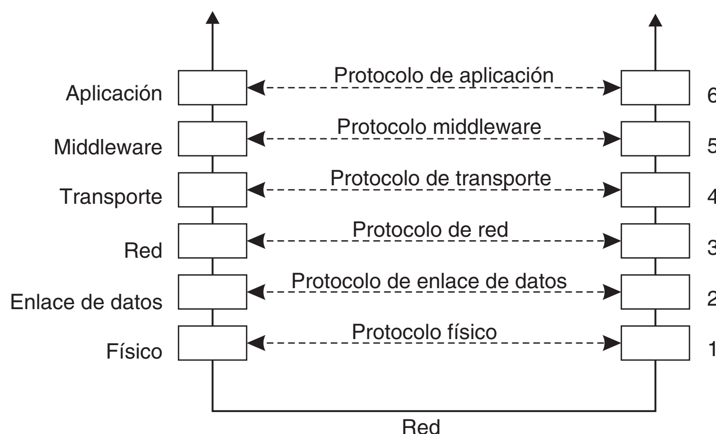


Figura 4-3. Modelo de referencia adaptado para comunicación en red.

Al adoptar este método de capas llegaremos a un modelo de referencia adaptado para comunicación, como ilustra la figura 4-3. En comparación con el modelo OSI, las capas de sesión y presentación se reemplazaron con una sola capa middleware que contiene protocolos de aplicación independientes. Estos protocolos no pertenecen a las capas más bajas que acabamos de explicar. Los

servicios de transporte originales también pueden ser ofrecidos como un servicio middleware, sin ser modificados. Este enfoque es de algún modo análogo a ofrecer UDP a nivel de transporte. De igual manera, los servicios de comunicación middleware pueden incluir servicios de paso de mensajes comparables con los que ofrece la capa de transporte.

En el resto de este capítulo, nos concentraremos en cuatro servicios de comunicación middleware de alto nivel: llamadas a procedimientos remotos, servicios de cola de mensajes, soporte para comunicación de medios continuos a través de flujos, y multitransmisión. Antes de hacerlo, hay otros criterios generales para distinguir comunicación middleware que veremos a continuación.

4.1.2 Tipos de comunicación

Para comprender las diversas alternativas en comunicación que ofrece el middleware a las aplicaciones, veremos al middleware como un servicio adicional en el cómputo cliente-servidor, tal como ilustra la figura 4-4. Por ejemplo, considere un sistema de correo electrónico. En principio, la parte central del sistema de entrega de correo puede considerarse como un servicio de comunicación middleware. Cada servidor ejecuta un agente usuario que permite a los usuarios redactar, enviar, y recibir correo electrónico. Un agente usuario de envío pasa tal correo al sistema de entrega de correo, esperando a su vez entregar el correo al destinatario en algún momento. De igual manera, el agente usuario presente del lado del destinatario se conecta al sistema de entrega de correo para ver si ha llegado algún correo. Si es así, los mensajes se transfieren al agente usuario de tal modo que puedan desplegarse y ser leídos por el usuario.

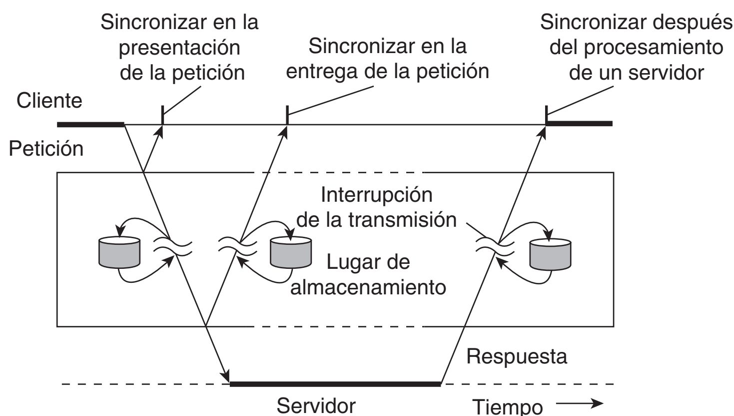


Figura 4-4. Perspectiva del middleware como un servicio intermediario (distribuido) al nivel de comunicación de aplicaciones.

Un sistema de correo electrónico es un ejemplo típico en el que la comunicación es persistente. Con **comunicación persistente**, el middleware de comunicación almacena el mensaje que ha sido presentado para transmitirse el tiempo que tome entregarlo al destinatario. En este caso, el middleware almacenará el mensaje en uno o varios de los lugares de almacenamiento que aparecen

en la figura 4-4. Como consecuencia, no resulta necesario que la aplicación remitente continúe con la ejecución después de presentar el mensaje. De igual manera, la aplicación destinataria no necesita estar en ejecución cuando el mensaje se presenta.

Por contraste, con la **comunicación transitoria**, el sistema de comunicación almacena un mensaje sólo mientras las aplicaciones remitente y destinataria se ejecutan. Más precisamente, en términos de la figura 4-4, el middleware no puede entregar un mensaje debido a una interrupción en la transmisión, o debido a que el destinatario no está activo de momento, y simplemente será descartado. En general, todos los servicios de comunicación al nivel de transporte sólo ofrecen comunicación transitoria. En este caso, el sistema de comunicación consiste en ruteadores tradicionales de almacenamiento y envío. Si un ruteador no puede entregar un mensaje al siguiente ruteador, o a la máquina de destino, simplemente se deshará del mensaje.

Además de ser persistente o transitoria, la comunicación también puede ser asíncrona o sincrónica. La característica principal de la **comunicación asíncrona** es que el remitente continúa inmediatamente después de que ha pasado su mensaje para transmisión. Esto significa que el mensaje es almacenado (de modo temporal) inmediatamente bajo la supervisión del middleware. Con la **comunicación sincrónica**, el remitente es bloqueado hasta que se sabe que su petición es aceptada. Básicamente existen tres casos en que puede ocurrir la sincronización. Primero, el remitente puede ser bloqueado hasta que el middleware notifica que va a encargarse de la transmisión de la petición. Segundo, el remitente puede sincronizarse hasta que su petición se ha entregado al destinatario. Tercero, la sincronización puede ocurrir al dejar que el remitente espere hasta que su petición se haya procesado por completo, es decir, hasta que el destinatario devuelva una respuesta.

En la práctica tienen lugar diversas combinaciones de persistencia y sincronización. Las más populares, son la persistencia en combinación con la sincronización por presentación de una petición —un esquema común para muchos sistemas de fila de mensajes—, las cuales explicaremos más adelante en este capítulo. De igual manera, la comunicación transitoria con sincronización luego de que una petición ha sido completamente procesada también se utiliza ampliamente. Este esquema corresponde a las llamadas a procedimientos remotos, que también explicaremos más adelante.

Además de la persistencia y la sincronización, también debemos diferenciar la comunicación discreta y por flujo. Los ejemplos que hemos dado hasta el momento, pertenecen a la categoría de comunicación discreta: las partes se comunican a través de mensajes, y cada mensaje forma una unidad completa de información. Por contraste, el flujo involucra el envío de varios mensajes, uno después de otro, y los mensajes se relacionan entre sí por el orden en que se envían, o porque existe una relación temporal. Más adelante retomaremos ampliamente la comunicación por flujo.

4.2 LLAMADAS A PROCEDIMIENTOS REMOTOS

Muchos sistemas distribuidos se han basado en el intercambio explícito de mensajes entre procesos. Sin embargo, los procedimientos `send` y `receive` no ocultan en absoluto la comunicación, lo cual es importante para lograr la transparencia en los sistemas distribuidos. Este problema, se conoce

desde hace mucho, pero poco se hizo hasta que un artículo escrito por Birrell y Nelson (1984) mostró una forma completamente diferente de manejar la comunicación. Aunque la idea es mantenerla simple (una vez que alguien ha pensado en ello), con frecuencia las implicaciones son sutiles. En esta sección examinaremos el concepto, su implementación, sus fortalezas, y sus debilidades.

En pocas palabras, lo que Birrell y Nelson sugirieron fue permitir que los programas llamaran a procedimientos ubicados en otras máquinas. Cuando un proceso de la máquina *A* llama a un procedimiento de la máquina *B*, el proceso que llama desde *A* se suspende, y la ejecución del procedimiento llamado ocurre en *B*. La información puede transportarse en los parámetros desde quien llama hasta el que es llamado, y puede regresar en el procedimiento resultante. Ningún mensaje de paso es visible para el programador. Este método se conoce como **llamada a procedimiento remoto**, o simplemente **RPC**.

Aunque la idea básica parece sencilla y elegante, existen problemas sutiles. Para empezar, debido a que el procedimiento que llama y el procedimiento llamado se ejecutan en máquinas diferentes, también se ejecutan en espacios de dirección diferentes, ello causa complicaciones. Los parámetros y resultados también deben pasarse, lo cual puede ser complicado, en especial si las máquinas no son idénticas. Por último, una o ambas máquinas pueden fallar, y cada una de las posibles fallas ocasiona diferentes problemas; aún así, la mayoría son manejables, y la RPC es una técnica ampliamente utilizada que muchos sistemas distribuidos soportan.

4.2.1 Operación básica RPC

Primero iniciamos con una explicación de las llamadas a procedimientos convencionales, y después explicamos cómo la propia llamada se divide en una parte cliente y una parte servidor y que cada parte se ejecuta en máquinas diferentes.

Llamada a un procedimiento convencional

Para comprender cómo funciona la RPC, es importante que primero comprendamos totalmente cómo funciona una llamada a un procedimiento convencional (es decir, en una sola máquina). Considere una llamada en C como

```
count = read(fd, buf, nbytes);
```

donde *fd* es un entero que indica un archivo, *buf* es un arreglo de caracteres en el que se leen los datos, y *nbytes* es otro entero que indica cuántos bytes leer. Si la llamada se hace desde el programa principal, la pila será como la que muestra la figura 4-5(a) antes de la llamada. Para realizar la llamada, el procedimiento que llama coloca los parámetros en la pila, en un orden del último al primero, como ilustra la figura 4-5(b). (La razón por la que los compiladores de C introduzcan los parámetros en orden inverso tiene que ver con *printf* —ya que al hacerlo, *printf* siempre puede localizar a su primer parámetro, la cadena de formato.) Después que el procedimiento read ha terminado su ejecución, coloca el valor de retorno en un registro, elimina la dirección de retorno, y devuelve el control al procedimiento que llama. Este último elimina entonces los parámetros de la pila y la devuelve al estado original que tenía antes de la llamada.

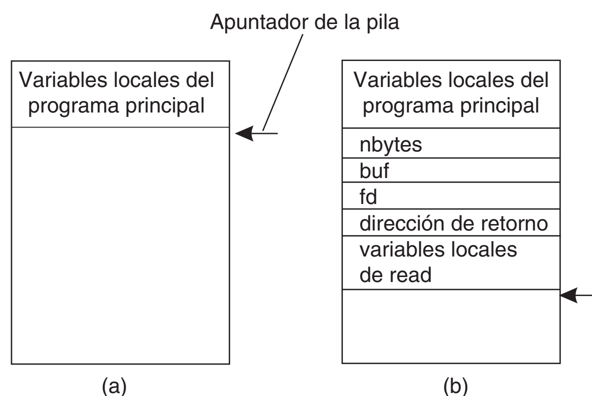


Figura 4-5. (a) Paso de parámetros en una llamada a un procedimiento local: la pila antes de la llamada a read. (b) La pila mientras el procedimiento llamado está activo.

Es importante observar algunas cosas. Por ejemplo, en C, los parámetros pueden ser **llamadas por valor** o **llamadas por referencia**. Un parámetro por valor, como *fd* o *nbytes*, simplemente se copia en la pila, según muestra la figura 4-5(b). Para el procedimiento llamado, un parámetro por valor es sólo una variable local inicializada. El procedimiento llamado puede modificarla, pero tales cambios no afectan el valor original del lado del que llama.

Un parámetro por referencia en C es un apuntador a una variable (es decir, la dirección de la variable), en lugar del valor de la variable. En la llamada a read, el segundo parámetro es un parámetro de referencia, ya que en C los arreglos siempre pasan por referencia. Lo que en realidad se introduce en la pila es la dirección del arreglo de caracteres. Cuando el procedimiento llamado utiliza este parámetro para almacenar algo en el arreglo de caracteres, *sí* modifica el arreglo en el procedimiento que llama. La diferencia entre una llamada por valor y una llamada por referencia es muy importante para una RPC, como veremos.

También existe otro mecanismo de paso de parámetros, aunque no se utiliza en C, denominado **llamada por copia-restauración**. Este mecanismo, consiste en hacer que el procedimiento que llama copie la variable en la pila, como en la llamada por valor, y posteriormente la vuelva a copiar después de la llamada, sobreescribiendo su valor original. Bajo casi todas las condiciones, esto logra exactamente el mismo efecto que una llamada por referencia, pero en algunas circunstancias —digamos, que el mismo parámetro se presente varias veces en la lista de parámetros— las semánticas son diferentes. El mecanismo de llamada por copia-restauración no se utiliza en muchos lenguajes.

La decisión sobre qué mecanismo de paso de parámetros utilizar, corresponde normalmente a los diseñadores del lenguaje, y es una propiedad fija del lenguaje. En algunas ocasiones esto depende del tipo de datos que se pasen. Por ejemplo, en C, los enteros y otros tipos escalares siempre se pasan por valor, mientras que los arreglos siempre se pasan por referencia. Algunos compiladores Ada utilizan la copia-restauración para parámetros **de entrada-salida**, pero otros, utilizan la llamada por referencia. La definición del lenguaje permite cualquier elección, lo cual hace que la semántica resulte un tanto confusa.

Resguardos del cliente y servidor

La idea tras la RPC es hacerla parecer como una llamada local. En otras palabras, deseamos que la RPC sea transparente —el procedimiento de llamada no debe advertir que el procedimiento llamado se ejecuta en una máquina diferente o viceversa—. Suponga que un programa necesita leer algunos datos desde un archivo. El programador coloca en el código una llamada a read para obtener los datos. En un sistema tradicional (de un solo procesador), la rutina read se extrae de la biblioteca mediante el enlazador y se inserta en el programa objeto. Éste es un procedimiento corto, y generalmente se implementa llamando a un sistema read equivalente. En otras palabras, el procedimiento read es un tipo de interfaz establecido entre el código de usuario y el sistema operativo local.

Aunque read hace una llamada al sistema, ésta se realiza de la manera usual, colocando los parámetros en la pila, como ilustra la figura 4-5(b). Por tanto, el programador no sabe que en realidad read está haciendo algo extraño.

La RPC logra su transparencia de manera análoga. Cuando read es realmente un procedimiento remoto (por ejemplo, uno que se ejecutará en un archivo de la máquina servidor), se coloca en la biblioteca una versión diferente de read, llamada **resguardo del cliente**. Como al original, a éste se le llama utilizando también la secuencia de la figura 4-5(b). Además, como el original, éste hace igualmente una llamada al sistema operativo local. Sólo a diferencia del original, éste no solicita datos al sistema operativo. En vez de eso, empaca los parámetros en un mensaje y solicita que el mensaje sea enviado al servidor, según muestra la figura 4-6. Si seguimos la llamada a send, la matriz cliente llama a receive y se bloquea a sí misma hasta que la respuesta regresa.

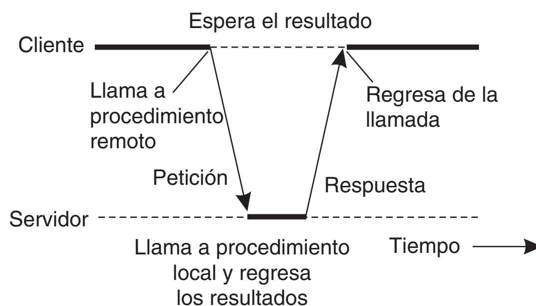


Figura 4-6. Principio de la RPC entre un programa cliente y servidor.

Cuando el mensaje llega al servidor, el sistema operativo lo pasa al **resguardo del servidor**. Un resguardo del servidor es el equivalente a un resguardo del cliente, del lado del servidor: una parte de código que transforma las peticiones entrantes a la red en llamadas a procedimientos locales. En general, la matriz servidor habrá llamado a receive y se habrá bloqueado esperando mensajes de entrada. El resguardo del servidor desempaca los parámetros del mensaje y después llama al procedimiento servidor de la manera usual (es decir, como en la figura 4-5). Desde el punto de

vista del servidor, es como pensar que está siendo llamado directamente por el cliente —los parámetros y las direcciones de retorno se encuentran en la pila a la que pertenecen, y nada parece inusual—. El servidor realiza su trabajo y después, del modo usual, regresa el resultado al procedimiento que llama. Por ejemplo, en el caso de read, el servidor llenará el bufer, señalado por el segundo parámetro, con los datos. Este bufer será interno para el resguardo del servidor.

Cuando el resguardo del servidor recupera el control después de que la llamada se ha completado, empaca el resultado (el bufer) en un mensaje y llama a send para devolverlo al cliente. Después de eso, la matriz servidor por lo general hace nuevamente una llamada a receive, para esperar la siguiente petición entrante.

Cuando el mensaje regresa a la máquina cliente, el sistema operativo del cliente ve que está dirigido hacia el proceso cliente (o en realidad a la parte del resguardo del cliente, pero el sistema operativo no puede advertir la diferencia). El mensaje se copia al bufer en espera y el proceso cliente se desbloquea. El resguardo del cliente inspecciona el mensaje, desempaca el resultado, lo copia para quien la llamó, y lo devuelve en la forma usual. Cuando quien llama obtiene el control siguiendo la llamada a read, todo lo que sabe, es que los datos están disponibles; no tiene idea de que el trabajo se hizo de manera remota en lugar de realizarlo el sistema operativo local.

Esta bendita ignorancia en la parte del cliente embellece todo el esquema. En cuanto a esto, se accede a los servicios remotos realizando llamadas a procedimientos ordinarios (es decir, locales), y no llamando a send y receive. Todos los detalles del paso de mensajes se ocultan en los dos procedimientos de bibliotecas, exactamente como se ocultan los detalles de las llamadas al sistema en bibliotecas tradicionales.

Para resumir, una llamada a un procedimiento remoto ocurre en los siguientes pasos:

1. El procedimiento cliente llama al resguardo del cliente de manera normal.
2. El resguardo del cliente construye un mensaje y llama al sistema operativo local.
3. El sistema operativo del cliente envía el mensaje al sistema operativo remoto.
4. El sistema operativo remoto da el mensaje al resguardo del servidor.
5. El resguardo del servidor desempaca los parámetros y llama al servidor.
6. El servidor realiza el trabajo y devuelve el resultado al resguardo.
7. El resguardo del servidor empaca el resultado en un mensaje y llama a su sistema operativo local.
8. El sistema operativo del servidor envía el mensaje al sistema operativo del cliente.
9. El sistema operativo del cliente da el mensaje al resguardo del cliente.
10. El resguardo desempaca el resultado y lo regresa al cliente.

El efecto neto de todos estos pasos es convertir la llamada local, mediante el procedimiento cliente hacia el resguardo del cliente, en una llamada local al procedimiento servidor sin que el cliente o el servidor adviertan los pasos intermedios o la existencia de la red.

4.2.2 Paso de parámetros

La función del resguardo del cliente es tomar sus parámetros, empacarlos en un mensaje, y enviarlos a la matriz servidor. Esto parece sencillo, pero no es tanto como puede parecer. En esta sección veremos algunas de las cuestiones relacionadas con el paso de parámetros en sistemas RPC.

Paso de parámetros de valor

Empacar parámetros en un mensaje se conoce como **ordenamiento de parámetros**. Como un ejemplo muy sencillo, considere un procedimiento remoto, add(i, j), que toma dos parámetros enteros, *i* y *j*, y devuelve la suma aritmética como resultado. (Como una cuestión práctica, uno no realizaría un procedimiento remoto tan simple debido a la sobrecarga, pero como ejemplo, lo haremos.) La llamada a add, aparece en el lado izquierdo (en el proceso cliente) de la figura 4-7. El resguardo del cliente toma sus dos parámetros y los coloca en un mensaje como se indica. También coloca el nombre o el número del procedimiento por llamar en el mensaje porque el servidor puede soportar muchas llamadas diferentes, y se le debe informar cuál llamada es requerida.

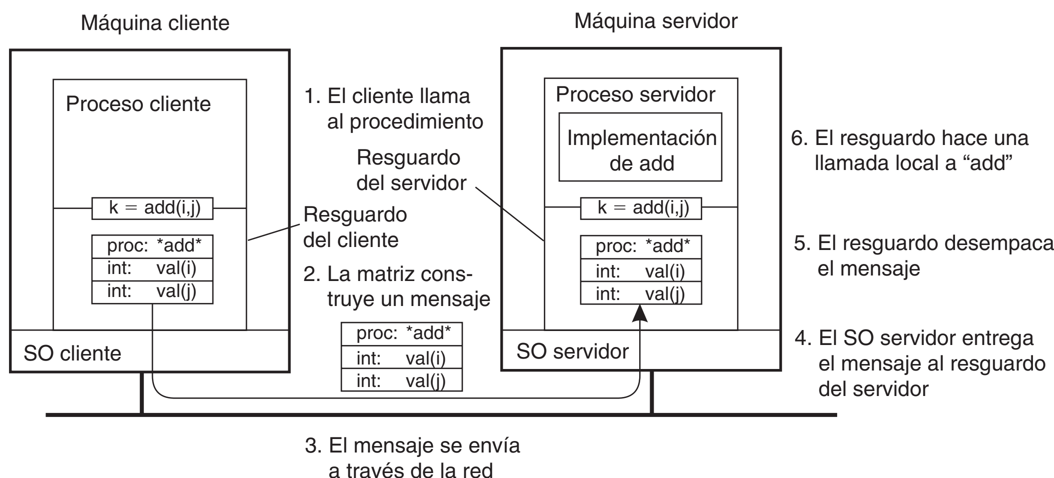


Figura 4-7. Pasos involucrados en un cálculo remoto a través de RPC.

Cuando el mensaje llega al servidor, el resguardo lo examina para ver qué procedimiento se necesita y realiza entonces la llamada adecuada. Si el servidor también da soporte a otros procedimientos remotos, el resguardo del servidor podría contar con una instrucción de cambio para seleccionar el procedimiento por llamar, de acuerdo con el primer campo del mensaje. La llamada real del resguardo al servidor parece la llamada cliente original, excepto por que los parámetros son variables inicializadas desde el mensaje entrante.

Cuando el servidor ha terminado, el resguardo del servidor obtiene nuevamente el control; toma el resultado enviado por el servidor y lo empaca en un mensaje. Este mensaje es enviado de

vuelta al resguardo del cliente, que lo desempaca para extraer el resultado y regresa el valor al procedimiento cliente en espera.

Mientras las máquinas cliente y servidor sean idénticas y todos los parámetros y resultados sean del tipo escalar, como enteros, caracteres y booleanos, este modelo funcionará bien. Sin embargo, en un sistema distribuido grande, es común encontrar varios tipos de máquinas. Con frecuencia, cada máquina tiene su propia representación para números, caracteres, y otros elementos de datos. Por ejemplo, las mainframes IBM utilizan el código de caracteres EBCDIC, mientras que las computadoras personales IBM emplean ASCII. En consecuencia, no es posible pasar un parámetro carácter de una PC IBM cliente a una mainframe IBM servidor utilizando el sencillo esquema de la figura 4-7: el servidor interpretará incorrectamente el carácter.

Problemas similares pueden suceder con la representación de enteros (complemento a uno *versus* complemento a dos) y de números de punto flotante. Además, un problema aún más molesto se presenta debido a que algunas máquinas, como la Intel Pentium, numeran sus bytes de derecha a izquierda mientras que otras, como las Sun SPARC, los numeran de forma inversa. El formato Intel se conoce como **little endian** y al formato SPARC se le llama **big endian**, por los políticos de *Los viajes de Gulliver* quienes fueron a la guerra para ver qué extremo de un huevo romper (Cohen, 1981). Como ejemplo, considere un procedimiento con dos parámetros, uno entero y una cadena de cuatro caracteres. Cada parámetro requiere una palabra de 32 bits. La figura 4-8(a) muestra cómo podría lucir la parte de parámetros de un mensaje construido por un resguardo cliente de una Intel Pentium. La primera palabra contiene el parámetro entero, en este caso 5, y la segunda contiene la cadena “JILL”.

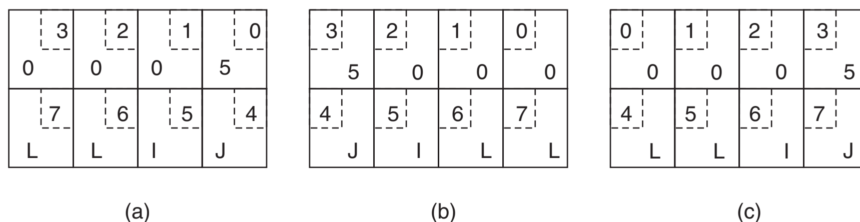


Figura 4-8. (a) El mensaje original en la Pentium. (b) El mensaje después de recibirlo la SPARC. (c) El mensaje después de ser invertido. Los pequeños números en los recuadros indican la dirección de cada byte.

Debido a que los mensajes se transfieren byte por byte (en realidad, bit por bit) a través de la red, el primer byte enviado es el primero en llegar. En la figura 4-8(b) mostramos cómo luciría el mensaje de la figura 4-8(a) de haber sido recibido por un SPARC, el cual numera sus bytes con el byte 0 a la izquierda (byte de mayor orden), en lugar de a la derecha (byte de menor orden), como lo hacen todos los chips de Intel. Cuando el resguardo del servidor lea los parámetros de las direcciones 0 y 4, respectivamente, encontrará un entero igual a 83 886 080 (5×2^{24}) y una cadena “JILL”.

Un método evidente, pero por desgracia incorrecto, es simplemente invertir los bytes de cada palabra después de recibirlos, vea la figura 4-8(c). Ahora el entero es 5 y la cadena es “LLIJ”. El

problema aquí es que los enteros son invertidos mediante un ordenamiento de bytes diferente, pero las cadenas no. Sin información adicional sobre lo que es una cadena y lo que es un entero, no hay manera de reparar el daño.

Paso de parámetros de referencia

Ahora nos enfrentamos a un problema difícil: ¿Cómo se pasan los apuntadores o, en general, las referencias? La respuesta es: sólo con mucha dificultad, si es posible. Recuerde que un apuntador tiene significado sólo dentro del espacio dirección del proceso en el que se utiliza. Si volvemos al ejemplo read explicado antes, cuando el segundo parámetro (la dirección del bufer) es 1000 en el cliente, no podemos simplemente pasar el número 1000 al servidor y esperar que funcione. La dirección 1000 en el servidor puede encontrarse a la mitad del texto del programa.

Una solución es simplemente prohibir los apuntadores y parámetros de referencia en general. Sin embargo, éstos son tan importantes que tal solución es bastante indeseable. De hecho, tampoco es necesaria. En el ejemplo de read, el resguardo del cliente sabe que el segundo parámetro apunta hacia un arreglo de caracteres. Por el momento, supongamos que también sabe qué tan grande es el arreglo. Entonces se vuelve evidente una estrategia: copiar el arreglo que hay en el mensaje y enviarlo al servidor. El resguardo del servidor puede llamar entonces al servidor con un apuntador hacia este arreglo, aunque este apuntador tenga un valor numérico diferente al del segundo parámetro de read. Los cambios que hace el servidor utilizando el apuntador (por ejemplo, almacenar datos en él) afectan directamente al bufer de mensaje dentro del resguardo del servidor. Cuando el servidor termina, el mensaje original puede enviarse de vuelta al resguardo del cliente, la cual después lo copia y regresa al cliente. En efecto, la llamada por referencia se reemplazó con una llamada por copia-restauración. Aunque esto no siempre es idéntico, con frecuencia resulta suficientemente bueno.

Una optimización duplica la eficiencia de este mecanismo. Si el resguardo sabe si el bufer es un parámetro de entrada o un parámetro de salida para el servidor, una de las copias puede eliminarse. Si el arreglo es entrada para el servidor (digamos, en una llamada a write), no es necesario volverlo a copiar. Si es salida, no es necesario mandarlo.

Como un comentario final, vale la pena observar que aunque ahora podemos manejar apuntadores hacia estructuras y arreglos sencillos, aún no podemos manejar el caso más general de un apuntador hacia una estructura de datos cualquiera, tal como un gráfico complejo. Algunos sistemas intentan lidiar con este caso pasando el apuntador al resguardo del servidor y generando código especial en el procedimiento servidor para utilizar apuntadores. Por ejemplo, una petición podría enviarse de vuelta al cliente para proporcionar los datos referenciados.

Especificación de parámetros y generación de resguardos

De lo que hemos explicado hasta el momento, queda claro que para ocultar una llamada a un procedimiento remoto es necesario que quien llama y quien es llamado coincidan en el formato de los mensajes que intercambian, y que sigan los mismos pasos cuando, digamos, pasan estructuras de datos complejas. En otras palabras, ambos lados de una RPC deben seguir el mismo protocolo, o la RPC no funcionará correctamente.

Como un sencillo ejemplo, consideremos el procedimiento de la figura 4-9(a). Tiene tres parámetros, un carácter, un número de punto flotante, y un arreglo de cinco enteros. Supongamos una palabra de cuatro bytes, el protocolo RPC podría presuponer que debemos transmitir un carácter en el byte situado más a la derecha de una palabra (y dejar vacíos los tres siguientes bytes), un flotante como una palabra completa, y un arreglo como un grupo de palabras igual a la longitud del arreglo, precedido por una palabra que dé la longitud, como ilustra la figura 4-9(b). Por tanto, dadas estas reglas, el resguardo del cliente para foobar sabe que debe utilizar el formato de la figura 4-9(b), y el resguardo del servidor sabe que los mensajes entrantes para foobar tendrán el formato de la figura 4-9(b).

```
foobar(char x; float y; int z[5])
{
  ....
}
```

(a)

Variables locales de foobar	
	x
y	
5	
z[0]	
z[1]	
z[2]	
z[3]	
z[4]	

(b)

Figura 4.9. (a) Procedimiento. (b) Mensaje correspondiente.

Definir el formato del mensaje es un aspecto de un protocolo RPC, pero no es suficiente. También necesitamos que el cliente y el servidor coincidan en la representación de estructuras de datos sencillas, tales como enteros, caracteres, booleanos, etc. Por ejemplo, el protocolo podría presuponer que los enteros se representen en complementos de dos, caracteres en Unicode de 16 bits, y flotantes en el formato estándar #754 de IEEE, almacenado todo en little endian. Con esta información adicional, los mensajes pueden interpretarse inequívocamente.

Con las reglas de codificación aseguradas hasta el último bit, sólo queda por hacer que quien llame y el llamado coincidan en el intercambio real de mensajes. Por ejemplo, podría decidirse utilizar un servicio de transporte orientado a conexión tal como TCP-IP. Una alternativa es utilizar un servicio de datagramas poco confiable y dejar que el cliente y el servidor implementen un esquema de control de errores como parte del protocolo RPC. En la práctica, existen diversas variantes.

Una vez que el protocolo RPC se ha definido completamente, es necesario implementar los resguardos del cliente y servidor. Afortunadamente, los resguardos para el mismo protocolo, pero para procedimientos diferentes, por lo general, difieren sólo en sus interfaces con las aplicaciones. Una interfaz consiste en una colección de procedimientos que pueden ser llamados por un cliente, y que son implementados por un servidor. Una interfaz está disponible normalmente en

el mismo lenguaje de programación en el que está escrito el cliente o el servidor (aunque estrictamente hablando, esto no es necesario). Para simplificar las cosas, las interfaces con frecuencia se especifican mediante un **Lenguaje de Definición de Interfaces** (**IDL**, por sus siglas en inglés). Una interfaz especificada en un IDL, posteriormente se compila en un resguardo del cliente y en un resguardo del servidor, junto con las interfaces adecuadas de tiempo de compilación o de tiempo de ejecución.

La práctica muestra que utilizar un lenguaje de definición de interfaces simplifica de manera importante las aplicaciones cliente-servidor basadas en RPC. Debido a que es fácil generar completamente resguardos del cliente y servidor, todos los sistemas middleware basados en RPC ofrecen un IDL para dar soporte al desarrollo de aplicaciones. En algunos casos, utilizar el IDL resulta obligatorio, como veremos en capítulos posteriores.

4.2.3 RPC asíncrona

Igual que en las llamadas a procedimientos convencionales, cuando un cliente llama a un procedimiento remoto, el cliente se bloqueará hasta que se devuelva una respuesta. El comportamiento estricto de solicitud-respuesta es innecesario cuando no hay un resultado por devolver, y sólo conduce a bloquear al cliente mientras pudo haber continuado y realizar trabajo útil antes de la petición de llamada a un procedimiento remoto. Ejemplos de que a menudo no hay necesidad de esperar una respuesta incluyen: la transferencia de dinero de una cuenta a otra, agregar entradas a una base de datos, iniciar servicios remotos, el procesamiento por lotes, etcétera.

Para soportar tales situaciones, los sistemas RPC pueden proporcionar facilidades para lo que conocemos como **RPC asíncronas**, mediante las cuales, un cliente continúa trabajando de inmediato después de emitir la petición RPC. Con RPC asíncronas, al momento en que recibe la petición de RPC, el servidor envía inmediatamente una respuesta hacia el cliente y después llama al procedimiento solicitado. La respuesta representa un acuse de recibo para el cliente de que el servidor va a procesar la RPC. El cliente continuará su trabajo sin mayor bloqueo tan pronto reciba el acuse del servidor. La figura 4-10(b) muestra cómo interactúan el cliente y el servidor en el caso de RPC asíncronas. Para comparar, la figura 4-10(a) muestra el comportamiento normal de una solicitud-respuesta.

Las RPC asíncronas, también pueden ser útiles cuando una respuesta está por devolverse pero el cliente no está preparado para esperarla, y mientras tanto, hace nada. Por ejemplo, un cliente podría querer buscar por anticipado las direcciones de red de un conjunto de servidores que espera contactar pronto. Mientras un servicio de asignación de nombres recopila esas direcciones, el cliente podría desear hacer otras cosas. En tales casos, tiene sentido organizar la comunicación entre el cliente y el servidor a través de dos RPC asíncronas, como ilustra la figura 4-11. El cliente primero llama al servidor y entrega una lista de nombres de máquinas que debe buscar, y continúa su trabajo cuando el servidor le entrega el acuse de recibo de esa lista. La segunda llamada la hace el servidor al cliente para entregarle las direcciones encontradas. Combinar dos RPC asíncronas algunas veces se conoce como **RPC síncrona diferida**.

Es importante observar que existen variantes de RPC asíncronas en las que el cliente continúa con su ejecución inmediatamente después de enviar la petición al servidor. En otras palabras, el

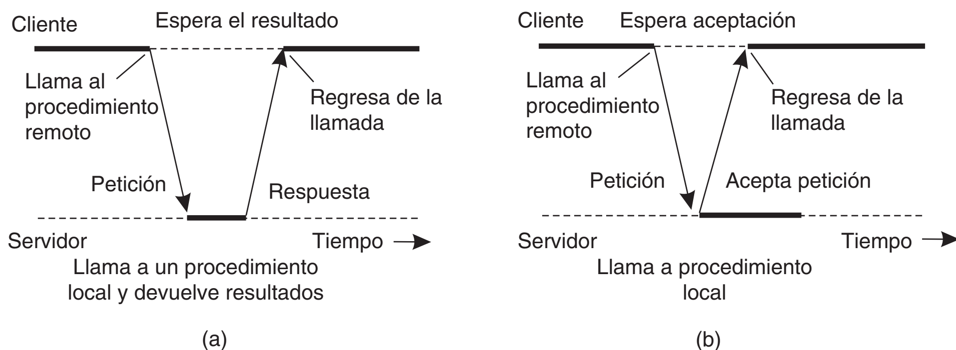


Figura 4-10. (a) Interacción entre cliente y servidor en una RPC tradicional. (b) Interacción utilizando RPC asíncronas.

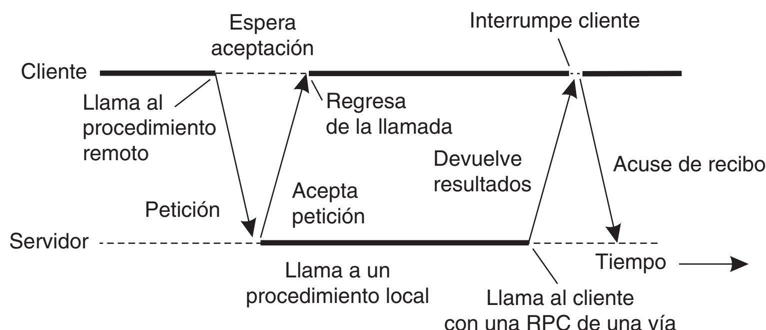


Figura 4-11. Cliente y servidor interactúan mediante dos RPC asíncronas.

cliente no espera un acuse del servidor por haber aceptado la petición. A una RPC como ésta la conocemos como **RPC de una vía**. El problema con este método es que cuando no se garantiza la confiabilidad, el cliente no puede saber con certeza si su petición será o no procesada. A estas cuestiones volveremos en el capítulo 8. De igual manera, en el caso de una RPC síncrona diferida, el cliente puede sondear al servidor para ver si los resultados están disponibles en lugar de dejar que el servidor le devuelva la llamada.

4.2.4 Ejemplo: DCE RPC

Las llamadas a procedimientos remotos se han adoptado ampliamente como la base del middleware y de los sistemas distribuidos en general. En esta sección, analizaremos con detalle un sistema específico RPC: el **Ambiente de Computación Distribuida** (**DCE**, por sus siglas en inglés), el cual fue desarrollado por Open Software Foundation (OSF), ahora conocida como The Open Group.

DCE RPC no es tan popular como otros sistemas RPC, por ejemplo, Sun RPC. Sin embargo, DCE RPC es representativo de otros sistemas RPC, y sus especificaciones se adoptaron en el sistema base de Microsoft para computación distribuida, DCOM (Eddon y Eddon, 1998). Iniciamos con una breve introducción a DCE, después de lo cual consideraremos las funciones principales de DCE RPC. El lector puede encontrar información técnica sobre cómo desarrollar aplicaciones basadas en RPC en Stevens (1999).

Introducción a DCE

DCE es un verdadero sistema middleware porque está diseñado para ejecutarse como una capa de abstracción entre sistemas operativos existentes (redes) y aplicaciones distribuidas. Inicialmente fue diseñado para UNIX, pero ahora se ha llevado a los sistemas operativos principales, incluso VMS y variantes de Windows, así como a sistemas operativos de escritorio. La idea es que el cliente pueda tomar una colección de máquinas existentes, agregar el software DCE, y después ejecutar aplicaciones distribuidas, todo sin molestar a las aplicaciones existentes (no distribuidas). Aunque la mayoría de los paquetes DCE se ejecutan en espacio de usuario, en algunas configuraciones es necesario agregar una pieza (parte del sistema distribuido de archivos) al kernel. El propio Open Group sólo vende código fuente, el cual los distribuidores integran a sus sistemas.

El modelo de programación subyacente a todo el DCE es el modelo cliente-servidor, el cual analizamos ampliamente en el capítulo anterior. Los procesos de usuario actúan como clientes para acceder a servicios remotos provistos mediante procesos servidor. Algunos de estos servicios son parte del propio DCE, pero otros, pertenecen a las aplicaciones y son escritos por los programadores de las aplicaciones. Toda comunicación entre clientes y servidores ocurre mediante RPC.

Existen diversos servicios que forman parte del propio DCE. El **servicio de archivos distribuidos** es un sistema mundial de archivos que proporciona una manera transparente de acceder a cualquier archivo del sistema en la misma forma; puede construirse en la capa superior de los sistemas de archivos nativos del servidor o utilizarse en lugar de estos sistemas. El **servicio de directorio** se utiliza para rastrear la ubicación de todos los recursos del sistema. Estos recursos incluyen máquinas, impresoras, servidores, datos, y mucho más, y pueden estar distribuidos geográficamente alrededor del mundo. El servicio de directorio permite a un proceso solicitar un recurso sin tener que preocuparse por dónde está, a menos que al proceso le importe. El **servicio de seguridad** permite que todos los recursos estén protegidos, de tal modo que el acceso pueda restringirse a personas sin autorización. Por último, el **servicio de tiempo distribuido** es un servicio que intenta mantener los relojes de las diferentes máquinas globalmente sincronizados. Como veremos en capítulos posteriores, tener cierta idea sobre el tiempo global facilita la garantía de la consistencia en un sistema distribuido.

Objetivos del DCE RPC

Los objetivos del sistema DCE RPC son relativamente tradicionales. El primero y más importante objetivo es que el sistema RPC haga posible que un cliente acceda a un servicio remoto mediante una simple llamada a un procedimiento local. Esta interfaz posibilita la escritura de programas cliente (es decir, aplicaciones) de manera sencilla y conocida para la mayoría de los programadores.

También facilita la ejecución de grandes volúmenes de código existente en un ambiente distribuido con pocos cambios, si los hay.

Depende del sistema RPC ocultar todos los detalles a los clientes, y hasta cierto punto, también a los servidores. Para comenzar, el sistema RPC puede localizar automáticamente el servidor correcto, y posteriormente establecer la comunicación entre el software cliente y servidor (en general, conocida como **vinculación**). También puede manejar el transporte de mensajes en ambas direcciones, fragmentarlos y rensamblarlos, según sea necesario (por ejemplo, si uno de los parámetros es un arreglo grande). Por último, el sistema RPC puede manejar automáticamente conversiones de tipos de datos entre el cliente y el servidor, incluso cuando se ejecutan en diferentes arquitecturas y tienen distintos ordenamientos de bytes.

Como una consecuencia de la habilidad del sistema RPC de ocultar los detalles, los clientes y servidores son muy independientes unos de otros. Un cliente puede escribirse en Java y un servidor en C, o viceversa. Un cliente y un servidor pueden ejecutarse en diferentes plataformas de hardware, y utilizar sistemas operativos distintos. También se da soporte a toda una variedad de protocolos de red y representaciones de datos, todo sin intervención alguna del cliente o del servidor.

Cómo escribir un cliente y un servidor

El sistema DCE RPC consiste en una cantidad de componentes que incluye lenguajes, bibliotecas, *demonios*, y programas de utilerías, entre otros; y juntos hacen posible escribir clientes y servidores. En esta sección describiremos las piezas y cómo juntarlas. La figura 4-12 muestra un resumen de todo el proceso de escribir y utilizar un RPC cliente y servidor.

En un sistema cliente-servidor, el pegamento que mantiene todo unido es la definición de interfaz, tal como se especifica en el **Lenguaje de Definición de Interfaces**, o **IDL** por sus siglas en inglés. El IDL permite implementar declaraciones de procedimientos en una forma muy parecida a los prototipos de funciones en ANSI C. Los archivos IDL también pueden contener definiciones de tipos, declaraciones de constantes, y otra información necesaria para organizar parámetros de manera correcta y desorganizar resultados. Idealmente, la definición de interfaz también debe contener una definición formal de lo que hacen los procedimientos, pero tal definición cae más allá del actual estado de cosas, por ello sólo define la sintaxis de las llamadas, no su semántica. Cuando mucho, quien escribe puede agregar algunos comentarios que describen lo que hacen los procedimientos.

Un elemento crucial en todo archivo IDL es un identificador globalmente único para la interfaz especificada. El cliente envía este identificador en el primer mensaje RPC y el servidor verifica que sea correcto. De esta manera, cuando inadvertidamente un cliente intenta vincularse con el servidor equivocado, o incluso con una versión anterior del servidor correcto, el servidor detectará el error y la vinculación no se llevará a cabo.

Las definiciones de interfaz y los identificadores únicos están muy relacionados en DCE. Tal como ilustra la figura 4-12, el primer paso para escribir una aplicación cliente-servidor es, en general, llamar al programa *uuidgen* para solicitarle que genere un archivo prototipo IDL que contenga un identificador de interfaz garantizado que no se utilice nuevamente en ninguna interfaz generada en cualquier otra parte por un *uuidgen*. La unicidad se garantiza codificando la ubicación y el tiempo

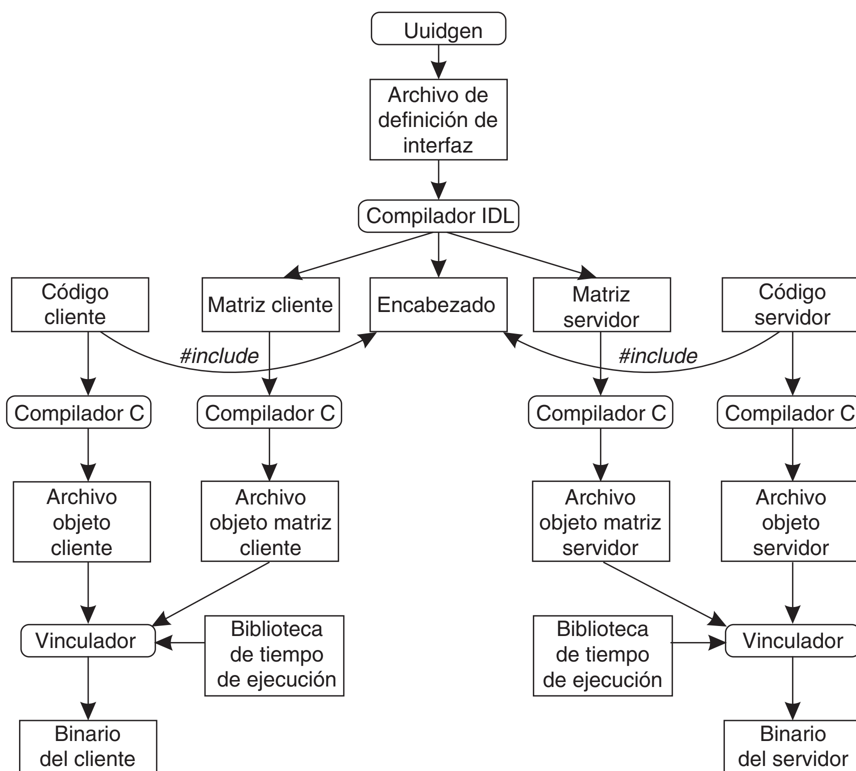


Figura 4-12. Pasos para escribir un cliente y un servidor en DCE RPC.

de creación. Esto consiste en un número binario de 128 bits representado en el archivo IDL como una cadena ASCII en hexadecimal.

El siguiente paso es editar el archivo IDL, escribiendo los nombres de los procedimientos remotos y sus parámetros. Vale la pena observar que la RPC no es totalmente transparente —por ejemplo, el cliente y el servidor no pueden compartir variables globales—, pero las reglas IDL hacen imposible expresar construcciones que no están soportadas.

Cuando el archivo IDL está completo, se llama al compilador IDL para que lo procese. La salida del compilador IDL consiste en tres archivos:

1. Un archivo de encabezado (por ejemplo, *interface.h*, en términos de C).
2. El resguardo del cliente.
3. El resguardo del servidor.

El archivo de encabezado contiene el identificador único, definiciones de tipos, definiciones de constantes, y prototipos de funciones. Este archivo debe incluirse (utilizando *#include*) tanto en el código cliente como en el código servidor. El resguardo del cliente contiene los procedimientos

reales que el programa cliente llamará. Estos procedimientos son los responsables de recopilar y empacar los parámetros en el mensaje de salida, y de llamar entonces al sistema en tiempo de ejecución para enviarlo. La matriz cliente maneja también el desempaque de la respuesta y la devolución de los valores al cliente. La matriz servidor contiene los procedimientos llamados por el sistema en tiempo de ejecución en la máquina servidor cuando llega un mensaje entrante. Estos procedimientos, a su vez, llaman a los procedimientos reales de servidor que hacen el trabajo.

El siguiente paso es para que la aplicación escritora escriba el código cliente y servidor. Después ambos se compilan como lo que son, dos procedimientos de resguardo. El código cliente resultante y los archivos objeto del resguardo del cliente se vinculan con la biblioteca en tiempo de ejecución para producir el binario ejecutable para el cliente. De manera similar, el código servidor y el resguardo del servidor se compilan y enlazan para producir el binario del servidor. En tiempo de ejecución, el cliente y el servidor se inician de tal modo que la aplicación en realidad también se ejecuta.

Vinculación de un cliente con un servidor

Con el propósito de permitir que un cliente llame a un servidor, es necesario que el servidor se registre y prepare para aceptar llamadas entrantes. El registro de un servidor hace posible que un cliente localice al servidor y se vincule con él. La localización de un servidor se hace en dos pasos:

1. Ubicación de la máquina servidor.
2. Ubicación del servidor (es decir, el proceso correcto) en esa máquina.

El segundo paso es de algún modo sutil. Básicamente, se trata de que, para comunicarse con un servidor, el cliente necesita conocer un **punto final** —ubicado en la máquina del servidor— al cual pueda enviar mensajes. El sistema operativo del servidor utiliza un punto final (también conocido generalmente como **puerto**) para diferenciar mensajes de entrada para diferentes procesos. En DCE, se mantiene una tabla de pares (*servidor, punto final*) en cada máquina servidor mediante un proceso llamado **demonio DCE**. Antes de estar disponible para peticiones de entrada, el servidor debe solicitar al sistema operativo un punto final; después registra este punto final con el demonio DCE que, a su vez, registra esta información (incluyendo los protocolos a los que se refiere el servidor) en la tabla de puntos finales para uso futuro.

El servidor también se registra con el servicio de directorio proporcionando la dirección de red de la máquina del servidor y un nombre con el que puede ser localizado. Vincular un cliente con un servidor se lleva a cabo tal como indica la figura 4-13.

Supongamos que el cliente quiere vincularse con un servidor de video que localmente es conocido con el nombre de */local/multimedia/video/movies*. El cliente pasa este nombre al servidor de directorio, el cual regresa la dirección de red de la máquina que ejecuta el servidor de video. El cliente se dirige entonces al demonio DCE localizado en esa máquina (que tiene un punto final bien conocido), y le solicita buscar el punto final del servidor de video en su tabla de puntos finales. Armado (el cliente) con esta información, la RPC ya puede llevarse a cabo. En RPC posteriores, esta búsqueda ya no será necesaria. El DCE también proporciona a los clientes la habilidad de realizar búsquedas más sofisticadas para los servidores adecuados cuando es necesario. La RPC segura también es una alternativa en la que la confidencialidad o la integridad de los datos resulta crucial.

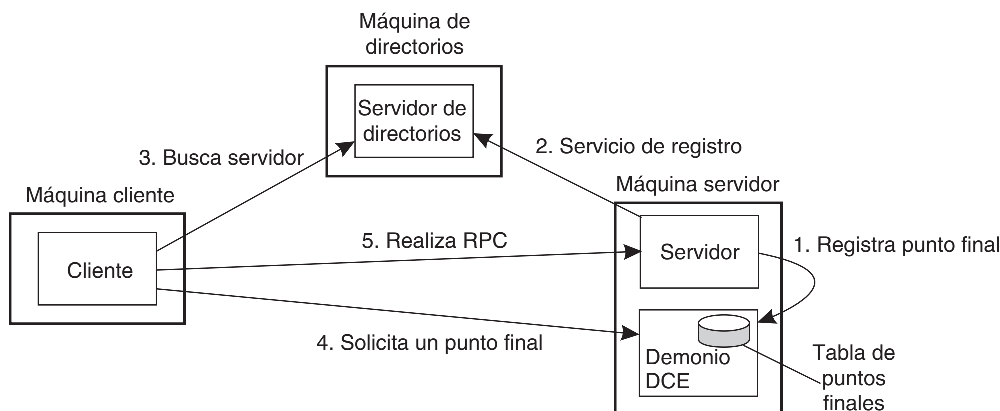


Figura 4-13. Vinculación cliente-servidor en DCE.

Cómo se realiza una RPC

La RPC real se lleva a cabo con transparencia y de la manera usual. El resguardo del cliente organiza los parámetros de la biblioteca de tiempo de ejecución para su transmisión utilizando el protocolo elegido en tiempo de vinculación. Cuando llega un mensaje del lado del servidor, se enruta hacia el servidor correcto de acuerdo con el punto final contenido en el mensaje de entrada. La biblioteca de tiempo de ejecución pasa el mensaje al resguardo del servidor, la cual desorganiza los parámetros y llama al servidor.

La DCE proporciona diversas alternativas semánticas. La alternativa predeterminada es **a lo más una vez**, en cuyo caso ninguna llamada se realiza más de una vez, incluso si el sistema se cae. En la práctica, esto significa que si un servidor se cae durante una RPC y después se recupera rápidamente, el cliente no repite la operación por temor a que se haya realizado una vez.

Como alternativa, es posible marcar un procedimiento remoto como **idempotente** (en el archivo IDL), en cuyo caso puede repetirse varias veces sin daño alguno. Por ejemplo, la lectura de un bloque específico de un archivo puede intentarse una y otra vez hasta lograrla. Cuando una RPC idempotente falla debido a la caída del servidor, el cliente puede esperar hasta que el servidor reinicie y luego intentar de nuevo. Otras semánticas también están disponibles (pero rara vez se utilizan), incluso para transmisión de la RPC a todas las máquinas de la red local. En el capítulo 8 retomaremos la semántica de la RPC, cuando expliquemos la RPC en presencia de fallas.

4.3 COMUNICACIÓN ORIENTADA A MENSAJES

En los sistemas distribuidos, las llamadas a procedimientos remotos y las invocaciones a objetos remotos contribuyen a ocultar la comunicación, es decir, mejoran la transparencia de acceso. Por desgracia, ningún mecanismo es siempre adecuado. En particular, cuando no es posible asumir que el lado receptor está en ejecución al momento en que se hace una petición, se necesitan servicios de

comunicación alternos. De igual manera, la naturaleza síncrona de las RPC, por la cual un cliente se bloquea hasta que su petición es procesada, algunas veces necesita remplazarse con algo más.

Ese algo más es la mensajería. En esta sección nos concentraremos en la comunicación orientada a mensajes en sistemas distribuidos. Primero veremos qué es exactamente el comportamiento síncrono y cuáles son sus implicaciones; después explicaremos los sistemas de mensajería que suponen que las partes se ejecutan en tiempo de comunicación; y por último, analizaremos los sistemas de colas de mensajes que permiten a los procesos intercambiar información, incluso si la otra parte no se ejecuta al momento en que se inicia la comunicación.

4.3.1 Comunicación transitoria orientada a mensajes

Muchos sistemas y aplicaciones distribuidos se construyen directamente sobre el sencillo modelo orientado a mensajes ofrecido por la capa de transporte. Para comprender esto y apreciar a los sistemas orientados a mensajes como parte de las soluciones middleware, primero explicaremos la mensajería a través del nivel de transporte mediante sockets.

Sockets Berkeley

Se ha puesto especial atención a la estandarización de la interfaz de la capa de transporte para permitir a los programadores utilizar la suite completa de los protocolos (de mensajería) mediante un simple conjunto de primitivas. Además, las interfaces estándar facilitan llevar una aplicación a una máquina diferente.

Como un ejemplo, explicaremos brevemente la **interfaz de sockets** tal como se presentó en la década de 1970 en Berkeley UNIX. Otra interfaz importante es la **XTI**, que significa **X/Open Transport Interface** (Interfaz Abierta de Transporte/X), antes llamada Interfaz de la Capa de Transporte (TLI), y desarrollada por AT&T. Las interfaces sockets y XTI son muy parecidas en sus modelos de programación de red, pero difieren en sus conjuntos de primitivas.

En cuanto a su concepto básico, un **socket** es un punto final de comunicación en el que una aplicación puede escribir información destinada a enviarse fuera de la red subyacente, y desde el cual puede leerse información entrante. Un socket forma una abstracción sobre el punto final real de comunicación, el cual utiliza el sistema operativo local para un protocolo de transporte específico. En el texto siguiente, nos concentraremos en las primitivas de sockets para TCP, las cuales aparecen en la figura 4-14.

Por lo general, los servidores ejecutan primero cuatro primitivas, normalmente en el orden dado. Cuando se llama a la primitiva socket, el que llama crea un nuevo punto final de comunicación para un protocolo de transporte específico. De manera interna, crear un punto final de comunicación significa que el sistema operativo local reserva recursos para alojar los mensajes enviados y recibidos por el protocolo especificado.

La primitiva bind asocia una dirección local con el socket creado recientemente. Por ejemplo, un servidor debe enlazar la dirección IP de su máquina con un número de puerto (probablemente muy conocido) para un socket. El enlace le indica al sistema operativo que el servidor sólo desea recibir mensajes en la dirección y el puerto especificados.

Primitiva	Significado
Socket	Crea un nuevo punto final de comunicación
Bind	Asocia una dirección local a un socket
Listen	Anuncia la conveniencia de aceptar conexiones
Accept	Bloquea a quien llama hasta que llega una petición de conexión
Connect	Activa el intento de establecer una conexión
Send	Envía algunos datos a través de la conexión
Receive	Recibe algunos datos a través de la conexión
Close	Libera la conexión

Figura 4-14. Primitivas de socket para TCP/IP.

La primitiva listen es llamada sólo en el caso de la comunicación orientada a conexión. Es una llamada que no bloquea y que permite al sistema operativo local reservar los búferes suficientes para un número máximo especificado de conexiones que quien llama está dispuesto a aceptar.

Una llamada a accept bloquea a quien llama hasta que llega una petición de conexión. Cuando llega una petición, el sistema operativo local crea un nuevo socket con las mismas propiedades del original, y lo devuelve a quien llama. Por ejemplo, este método permitirá al servidor generar un proceso que posteriormente manejará la comunicación real a través de la nueva conexión. Mientras tanto, el servidor puede volver y esperar otra petición de conexión con el socket original.

Ahora veamos del lado del cliente. Aquí también es necesario crear primero un socket mediante la primitiva socket, pero no es necesario enlazar explícitamente al socket con la dirección local ya que el sistema operativo puede destinar dinámicamente un puerto cuando se establece la conexión. La primitiva connect requiere que quien llama especifique la dirección al nivel de transporte a la que se enviará la petición de conexión. El cliente es bloqueado hasta que se establece exitosamente una conexión, después de lo cual ambos lados pueden iniciar con el intercambio de información a través de las primitivas send y receive. Por último, cerrar una conexión resulta simétrico cuando se utilizan sockets, y se logra haciendo que tanto el cliente como el servidor llamen a la primitiva close. La figura 4-15 muestra el patrón general que sigue un cliente y un servidor para implementar la comunicación orientada a conexiones mediante sockets. En Stevens (1998) encontrará todos los detalles necesarios sobre programación de redes mediante sockets y otras interfaces en un ambiente UNIX.

La interfaz de paso de mensajes (MPI)

Con la llegada de multicomputadoras de alto rendimiento, los desarrolladores han buscado primitivas orientadas a mensajes que les permitan escribir fácilmente aplicaciones altamente eficientes. Esto significa que las primitivas deben estar en un nivel de abstracción conveniente (para un desarrollo sencillo de aplicaciones), y que su implementación sólo implique una sobrecarga mínima. Se estimó

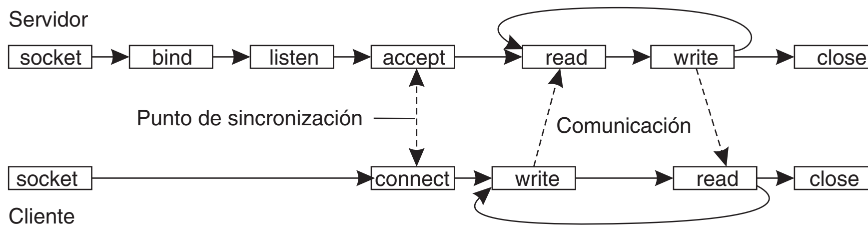


Figura 4-15. Patrón de comunicación orientada a conexiones mediante sockets.

que los sockets eran insuficientes por dos razones. Primero, se encontraban en el nivel de abstracción erróneo con el solo soporte de las primitivas simples para enviar y recibir. Segundo, los sockets se habían diseñado para comunicación a través de redes mediante protocolos de pilas de propósito general, como TCP/IP. Los sockets no se consideraron adecuados para los protocolos propietarios desarrollados para redes de interconexión de alta velocidad, tales como las utilizadas en clusteres de servidores de alto rendimiento. Aquellos protocolos requerían una interfaz que pudiese manejar características más avanzadas, como diferentes formas de utilizar el bufer y de sincronización.

El resultado fue que la mayoría de las redes de interconexión y multicomputadoras de alto rendimiento se pusieron en bibliotecas de comunicación propietarias. Estas bibliotecas ofrecían primitivas de comunicación valiosas, de alto nivel, y eficientes en general. Desde luego, todas las bibliotecas eran mutuamente incompatibles, por lo que los desarrolladores de aplicaciones tuvieron entonces un problema de portabilidad.

La necesidad de lograr independencia del hardware y de la plataforma derivó en algún momento en la definición de un estándar para el paso de mensajes, conocido simplemente como **Interfaz de Paso de Mensajes** o (**MPI**, por sus siglas en inglés). La MPI está diseñada para aplicaciones paralelas, y como tal, fue confeccionada para comunicación transitoria. Utiliza directamente la red subyacente. Además, asume que fallas serias, tal como caídas de procesos o particiones de red, son fatales y no requieren recuperación automática.

La MPI asume que la comunicación ocurre dentro de un grupo conocido de procesos. A cada grupo se le asigna un identificador. Dentro de un grupo, a cada proceso también se le asigna un identificador (local). Por lo tanto, un par (*IDgrupo*, *IDproceso*) únicamente identifica la fuente o el destino de un mensaje, y se utiliza en lugar de una dirección al nivel de transporte. Puede haber diversos grupos de procesos sobrepuestos involucrados en un cálculo, y que todos estén en ejecución al mismo tiempo.

En la parte central de la MPI se encuentran las primitivas de mensajería para soportar la comunicación transitoria, de las cuales las más conocidas aparecen en la figura 4-16.

La comunicación asíncrona transitoria se soporta mediante la primitiva MPI_bsend. El remitente presenta un mensaje para su transmisión, el cual, generalmente se copia primero en un bufer local de la MPI del sistema en ejecución. Cuando se ha copiado el mensaje, el remitente continúa. La MPI del sistema en ejecución eliminará el mensaje de su búfer local y se encargará de la transmisión tan pronto como un destinatario haya llamado a una primitiva de recepción.

Primitiva	Significado
MPI_bsend	Adjunta un mensaje de salida a un búfer local de envío
MPI_send	Envía un mensaje y espera hasta que es copiado en un búfer local o remoto
MPI_ssend	Envía un mensaje hasta que comienza la recepción
MPI_sendrecv	Envía un mensaje y espera una respuesta
MPI_isend	Pasa una referencia a un mensaje de salida, y continúa
MPI_issend	Pasa una referencia a un mensaje de salida, y espera hasta que inicia la recepción
MPI_recv	Recibe un mensaje; bloquea si no hay alguno
MPI_irecv	Verifica si hay algún mensaje de entrada, pero no bloquea

Figura 4-16. Algunas de las primitivas para paso de mensajes más conocidas de la MPI.

También hay una operación de envío que causa bloqueo, conocida como MPI_send, cuya semántica depende de la implementación. La primitiva MPI_send puede bloquear al que llama hasta que el mensaje especificado se haya copiado en la MPI del sistema en ejecución del lado del remitente, o hasta que el destinatario haya iniciado una operación de recepción. La comunicación síncrona mediante la que el remitente se bloquea hasta que su petición es aceptada para un mayor procesamiento está disponible a través de la primitiva MPI_ssend. Por último, también es soportada la forma más fuerte de comunicación síncrona: cuando un remitente llama a MPI_sendrecv, ésta envía una petición al destinatario y se bloquea hasta que le devuelve una respuesta. Básicamente, esta primitiva corresponde a una RPC normal.

Tanto MPI_send como MPI_ssend tienen variantes que evitan la copia de mensajes desde buferes de usuarios a buferes internos y a la MPI del sistema en ejecución. Estas variantes corresponden a una forma de comunicación asíncrona. Con MPI_isend, un remitente pasa un apuntador al mensaje, y después la MPI del sistema en ejecución se ocupa de la comunicación. El remitente continúa de inmediato. Para prevenir la sobreescritura en el mensaje antes de que la comunicación se complete, la MPI ofrece primitivas para verificar que se complete, o incluso que se bloquee si es necesario. Así como con MPI_send, no está especificado si el mensaje se ha transferido hacia el destinatario o si simplemente lo ha copiado la MPI local del sistema en ejecución hacia un bufer interno.

De igual manera, con MPI_issend, también un remitente pasa únicamente un apuntador a la MPI del sistema en ejecución. Cuando el sistema en ejecución indica que ha procesado el mensaje, el remitente está seguro de que el destinatario ha aceptado el mensaje y trabajará en él.

La operación MPI_recv es llamada para recibir un mensaje; ésta bloquea a quien llama hasta que llega un mensaje. También hay una variante asíncrona, conocida como MPI_irecv, mediante la cual un destinatario indica que está preparado para aceptar un mensaje. El destinatario puede verificar si en realidad ha llegado un mensaje, o se bloquea hasta que llega alguno.

La semántica de las primitivas de comunicación MPI no siempre es directa, y en ocasiones es posible intercambiar diferentes primitivas sin afectar la integridad de un programa. La razón oficial por la que se soportan muchas formas diferentes de comunicación es que proporciona a quienes implementan sistemas MPI suficientes posibilidades para optimizar el rendimiento. Los cínicos podrían decir, que el comité no pudo superar su mente colectiva, por lo que agregaron todo. La MPI se diseñó para aplicaciones paralelas de alto rendimiento, ello facilita comprender su diversidad en diferentes primitivas de comunicación.

En Gropp y colaboradores (1998b) puede encontrarse más sobre MPI. La referencia completa en la que se explican con detalle más de 100 funciones de MPI es Snir y colaboradores (1998), y Gropp y colaboradores (1998a).

4.3.2 Comunicación persistente orientada a mensajes

Ahora, explicaremos una clase importante de servicios middleware orientados a mensajes, generalmente conocidos como **sistemas de colas de mensajes**, o simplemente **Middleware Orientado a Mensajes (MOM**, por sus siglas en inglés). Los sistemas de colas de mensajes proporcionan un amplio soporte para comunicación asíncrona persistente. La esencia de estos sistemas es que ofrecen capacidad de almacenamiento de término medio para mensajes, sin la necesidad de que el remitente o el destinatario estén activos durante la transmisión del mensaje. Una diferencia importante con los sockets Berkeley y la MPI, es que los sistemas de colas de mensajes están dirigidos al soporte de transferencias de mensajes que toman minutos en lugar de segundos o milisegundos. Primero explicaremos un método general de los sistemas de colas de mensajes, y concluiremos esta sección comparándolos con sistemas más tradicionales tales como los sistemas de correo electrónico de internet.

Modelo de colas de mensajes

La idea básica detrás de un sistema de colas de mensajes, es que las aplicaciones se comunican insertando mensajes en colas específicas. Estos mensajes son reenviados a una serie de servidores de comunicación y en algún momento se entregan en su destino, incluso si éste no estaba disponible cuando se envió el mensaje. En la práctica, la mayoría de los servidores de comunicación están directamente conectados uno con otro. En otras palabras, por lo general, un mensaje se transfiere directamente hacia un servidor destino. En principio, cada aplicación tiene su propia cola privada a la que otras aplicaciones pueden enviar mensajes. Una cola puede ser leída sólo por su aplicación asociada, pero también es posible que varias aplicaciones compartan una sola cola.

Un aspecto importante de los sistemas de colas de mensajes es que, por lo general, al remitente sólo se le garantiza que su mensaje se insertará en algún momento en la cola del destinatario. No se dan garantías sobre cuándo, o de si el mensaje en realidad será leído, lo cual es completamente definido por el comportamiento del destinatario.

Estas semánticas permiten la comunicación muy poco acoplada en tiempo. Entonces, no hay necesidad de que el destinatario se encuentre en ejecución cuando un mensaje se envía a su cola. De igual manera, no hay necesidad de que el remitente se encuentre en ejecución al momento en que su mensaje es recogido por el destinatario. El remitente y el destinatario pueden ejecutarse com-

pletamente independientes uno de otro. De hecho, una vez que se ha depositado un mensaje en una cola, permanecerá ahí hasta que sea eliminado, sin tomar en cuenta si el remitente o el destinatario se encuentran en ejecución. Esto nos proporciona cuatro combinaciones con respecto al modo de ejecución del remitente y del destinatario, como ilustra la figura 4-17.

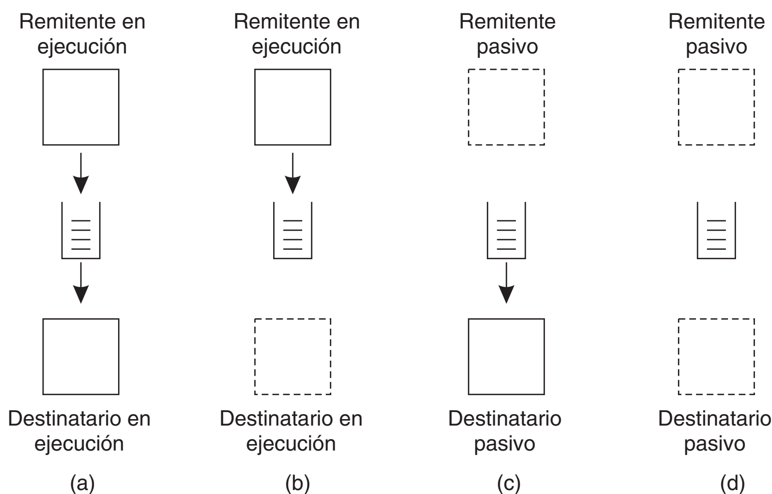


Figura 4-17. Cuatro combinaciones de comunicaciones muy poco acopladas mediante el uso de colas.

En la figura 4-17(a), tanto el remitente como el destinatario están en ejecución durante toda la transmisión de un mensaje. En la figura 4-17(b), sólo el remitente está en ejecución mientras el destinatario se encuentra pasivo, es decir, en un estado en el que no es posible la entrega del mensaje. Sin embargo, el remitente puede enviar mensajes. La combinación de un remitente pasivo y un destinatario en ejecución aparece en la figura 4-17(c). En este caso, el destinatario puede leer mensajes que le fueron enviados, pero no es necesario que sus respectivos remitentes se encuentren en ejecución. Por último, en la figura 4-17(d), vemos la situación en que el sistema almacena (y posiblemente transmite) mensajes, incluso mientras el remitente y el destinatarios se encuentran pasivos.

En principio, los mensajes pueden contener cualquier información. El único aspecto importante a cubrir desde la perspectiva del middleware es que los mensajes se direccionen adecuadamente. En la práctica, el direccionamiento se hace proporcionando un nombre único de la cola de destino. En algunos casos el tamaño del mensaje puede limitarse, aunque también es posible que el sistema subyacente se encargue de fragmentar y ensamblar grandes mensajes de manera completamente transparente para las aplicaciones. Un efecto de este método, es que la interfaz básica ofrecida a las aplicaciones puede ser extremadamente simple, como indica la figura 4-18.

Un remitente llama a la primitiva put para pasar un mensaje al sistema subyacente que va a adjuntarse a la cola especificada. Como explicamos, ésta es una llamada que no bloquea. La primitiva get es una llamada bloqueadora mediante la cual un proceso autorizado puede eliminar el

Primitiva	Significado
Put	Adjunta un mensaje a una cola especificada
Get	Bloquea hasta que la cola especificada no esté vacía y elimina el primer mensaje
Poll	Verifica una cola especificada de mensajes y elimina el primero. Nunca bloquea
Notify	Instala un manejador al que se llamará cuando un mensaje se coloque en la cola especificada

Figura 4-18. Interfaz básica de una cola en un sistema de colas de mensajes.

mensaje pendiente más antiguo de la cola especificada. El proceso es bloqueado sólo si la cola está vacía. Variantes de esta llamada permiten la búsqueda de un mensaje específico en la cola, por ejemplo, mediante una prioridad o un patrón de coincidencias. La variante que no bloquea está dada por la primitiva poll. Si la cola está vacía, o si un mensaje específico no pudo encontrarse, el proceso de llamada simplemente continúa.

Por último, la mayoría de los sistemas de colas también permiten desarrollar un proceso para instalar un manejador como una función de *devolución de llamada*, la cual, es automáticamente invocada cuando un mensaje se coloca en la cola. Las devoluciones de llamada también pueden utilizarse para iniciar automáticamente un proceso que traerá mensajes de la cola si ningún proceso se encuentra en ejecución. Este método se implementa con frecuencia mediante un demonio del lado del destinatario, quien continuamente monitorea los mensajes entrantes a la cola y los maneja según corresponda.

Arquitectura general de un sistema de colas de mensajes

Ahora veamos cómo luce en general un sistema de colas de mensajes. Una de las primeras restricciones que hacemos es que los mensajes sólo pueden colocarse en colas que sean *locales* para el remitente, es decir, colas en la misma máquina, o nada peor que en una máquina cercana tal como en la misma LAN que puede ser alcanzada eficientemente a través de una RPC. A tal cola se le conoce como **cola fuente**. De igual manera, los mensajes sólo pueden leerse desde colas locales. Sin embargo, un mensaje colocado en una cola contendrá la especificación de una **cola de destino** a la que debe ser transferido. Es responsabilidad del sistema de colas de mensajes proporcionar colas a los remitentes y destinatarios, y encargarse de que los mensajes se transfieran desde su cola fuente hasta su cola destino.

Es importante advertir que la colección de colas se distribuye a través de diversas máquinas. En consecuencia, para que un sistema de colas de mensajes transfiera mensajes, es necesario que mantenga un mapa de las colas para localización en la red. En la práctica, esto significa que debe mantener una base de datos (probablemente distribuida) de los **nombres de las colas** para localizarlas en la red, como ilustra la figura 4-19. Observe que tal mapeo es completamente análogo al uso del Servicio de Nombres de Dominio (DNS, por sus siglas en inglés) para correo electrónico en internet. Por ejemplo, cuando se envía un mensaje a la dirección lógica de *correo steen@cs.vu.nl*, el sistema de correo consultará al DNS para localizar la dirección de *red* (es decir, la IP) del servidor de correo del destinatario para usarlo en la transferencia real del mensaje.

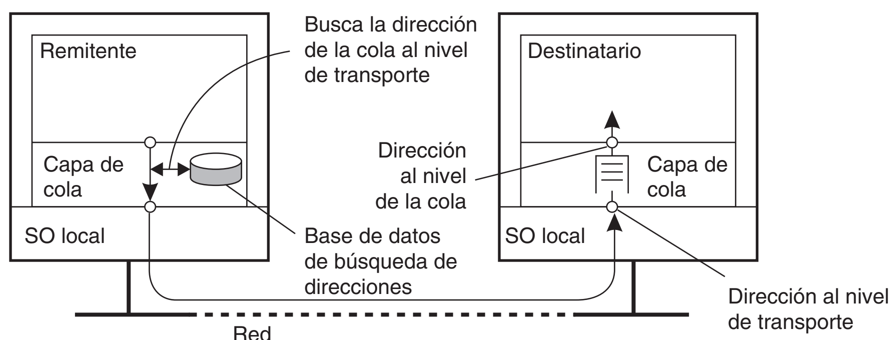


Figura 4-19. Relación entre el direccionamiento al nivel de cola y el direccionamiento al nivel de red.

Las colas son manejadas por **administradores de colas**. Por lo general, un administrador de colas interactúa directamente con la aplicación que está enviando o recibiendo un mensaje. Sin embargo, también hay administradores especiales de colas que operan como ruteadores, o **retransmisores**: éstos retransmiten mensajes entrantes hacia otros administradores de colas. De esta manera, un sistema de colas de mensajes puede convertirse en toda una **red sobrepuesta**, al nivel de aplicaciones, en el nivel superior de una red de computadoras existente. Este método es similar a la construcción de Mbone en internet, donde los procesos ordinarios de usuario se configuraron como ruteadores de multitransmisión. Como podemos ver, la multitransmisión a través de redes sobrepuestas aún es importante, y la explicaremos más adelante en este capítulo.

Los retransmisores pueden resultar convenientes por varias razones. Por ejemplo, en muchos sistemas de colas de mensajes no hay un servicio de asignación de nombres disponible y que pueda mantener dinámicamente mapas para localización de colas. En cambio, la topología de la red de colas es estática, y cada administrador de colas necesita una copia de dichos mapas. No hay necesidad de decir que en los grandes sistemas de colas este método, puede derivar en problemas de administración de red.

Una solución es utilizar algunos ruteadores que conozcan la topología de la red. Cuando un remitente *A* coloca un mensaje en su cola para el destinatario *B*, ese mensaje se transfiere primero al ruteador más cercano, digamos *R1*, como ilustra la figura 4-20. En ese punto, el ruteador sabe qué hacer con el mensaje y lo reenvía en la dirección de *B*. Por ejemplo, *R1* puede deducir, a partir del nombre de *B*, que el mensaje debe reenviarse al ruteador *R2*. De este modo, sólo es necesario actualizar los ruteadores cuando las colas se agregan o eliminan, mientras que todos los demás administradores de colas sólo necesitan saber en dónde se encuentra el ruteador más cercano.

Así, los retransmisores pueden ayudar a construir sistemas escalables de colas de mensajes. Sin embargo, cuando las redes de colas crecen, resulta claro que la configuración manual de las redes se volverá rápidamente no administrable. La única solución es adoptar esquemas de enrutamiento dinámico tal como se hace con redes de computadoras. Con respecto a esto, de alguna manera resulta sorprendente que tales soluciones aún no se integran en algunos de los sistemas de colas de mensajes más populares.

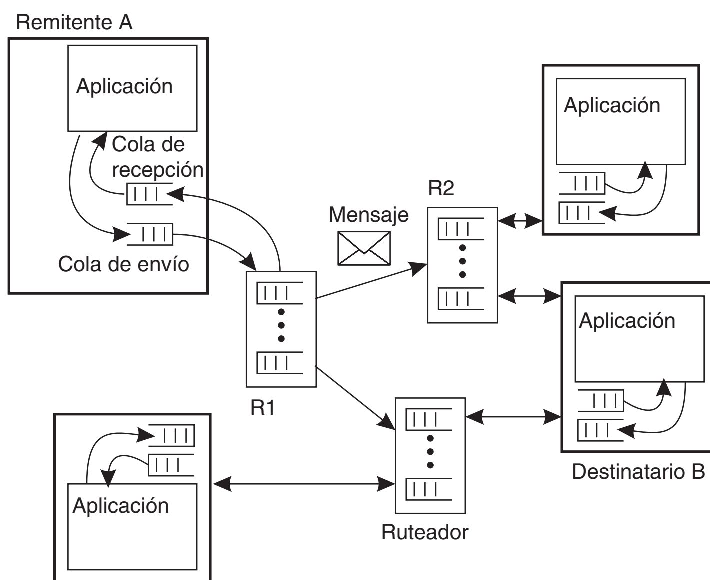


Figura 4-20. Organización general de un sistema de colas de mensajes con ruteadores.

Otra razón por la cual se utilizan retransmisores es que permiten el procesamiento secundario de mensajes. Por ejemplo, los mensajes pueden necesitar un registro por razones de seguridad o de tolerancia a fallas. Una forma especial de retransmisor, que explicaremos en la siguiente sección, es una que actúa como puerta de enlace, cambiando los mensajes a un formato que el destinatario puede entender.

Por último, los retransmisores pueden utilizarse con propósitos de multitransmisión. En ese caso, un mensaje entrante simplemente se coloca en cada cola de envío.

Agentes de mensajes

Un área de aplicación importante de los sistemas de colas de mensajes, es la integración de aplicaciones nuevas y existentes en un solo sistema de información distribuido y coherente. La integración requiere que las aplicaciones puedan comprender los mensajes recibidos. En la práctica, esto requiere que el remitente tenga sus mensajes de salida en el mismo formato que el destinatario.

El problema con este método es que cada vez que una aplicación agregada al sistema necesite un formato diferente de mensajes, cada remitente potencial tendrá que ajustarse para poder producir ese formato.

Una alternativa es coincidir con un formato común de mensajes, como se hace con los protocolos tradicionales de red. Por desgracia, este método generalmente no funciona con los sistemas de colas de mensajes. El problema subyacente es el nivel de abstracción en el que estos sistemas

operan. Un formato común de mensajes tiene sentido sólo si la colección que utiliza dicho formato tiene en realidad suficientes cosas en común. Si la colección de aplicaciones que forman un sistema de información distribuido es muy diversa (y con frecuencia es así), entonces el mejor formato común puede ser no más que una secuencia de bytes.

Aunque se han definido algunos formatos comunes de mensajes para dominios de aplicaciones específicas, el método general es aprender a vivir con formatos diferentes, e intentar proveer los medios para que las conversiones sean tan sencillas como sea posible. En sistemas de colas de mensajes, las conversiones son manejadas por nodos especiales de una red de colas conocidos como **agentes de mensajes**. Un agente de mensaje actúa como una puerta de enlace al nivel de aplicaciones en un sistema de colas de mensajes. Su propósito principal es convertir los mensajes entrantes de tal manera que la aplicación de destino pueda comprenderlos. Observe que para un sistema de colas de mensajes, un agente de mensaje es sólo otra aplicación, como indica la figura 4-21. En otras palabras, un agente de mensaje no se considera, por lo general, parte integral del sistema de colas.

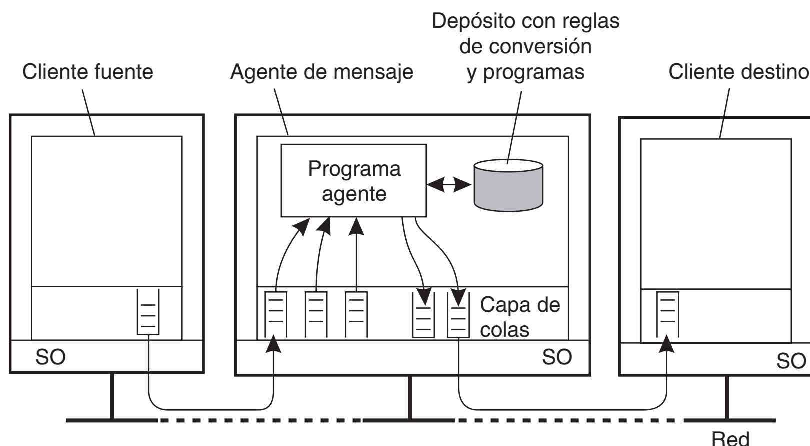


Figura 4-21. Organización general de un agente de mensaje en un sistema de colas de mensajes.

Un agente de mensajes puede ser tan sencillo como un reformateador de mensajes. Por ejemplo, supongamos que un mensaje entrante contiene una tabla de una base de datos en la que los registros están separados por un delimitador especial de *fin de registro*, y que los campos localizados dentro de un registro tienen una longitud fija conocida. Si la aplicación destino espera un delimitador diferente entre registros, y también espera que los campos tengan longitudes variables, se puede utilizar un agente de mensajes para convertirlos al formato esperado por el destinatario.

En una configuración avanzada, un agente de mensaje puede actuar como una puerta de enlace al nivel de aplicaciones similar a la puerta que maneja la conversión entre dos aplicaciones de bases de datos diferentes. En tales casos, no es posible garantizar que toda la información

contenida en el mensaje de entrada pueda transformarse realmente en algo adecuado para el mensaje de salida.

Sin embargo, es más común utilizar un agente de mensajes para la **integración de aplicaciones empresariales** (**EAI**, por sus siglas en inglés), como explicamos en el capítulo 1. En este caso, en lugar de (sólo) convertir mensajes, un agente es responsable de hacer coincidir aplicaciones basadas en los mensajes que se están intercambiando. En tal modelo, llamado de **publicación-suscripción**, las aplicaciones envían mensajes en la forma de *publicaciones*. En particular, pueden publicar un mensaje sobre el tema *X*, el cual después se envía al agente. Las aplicaciones que han expresado su interés por el tema *X*, es decir, que han *suscrito* dichos mensajes, recibirán entonces estos mensajes desde el agente. También hay formas más avanzadas de mediación, pero postergaremos su explicación hasta el capítulo 13.

En el núcleo de un agente de mensaje se encuentra un depósito de reglas y programas que pueden transformar un mensaje del tipo *T1* en uno del tipo *T2*. El problema es definir las reglas y desarrollar los programas. La mayoría de los productos de agentes de mensajes, vienen con herramientas sofisticadas de desarrollo, pero el resultado final es que el depósito debe ser llenado por expertos. Aquí tenemos un ejemplo perfecto, donde productos comerciales afirman proveer, a menudo engañosamente, "inteligencia", cuando en realidad la inteligencia radica en las cabezas de los expertos.

Nota sobre sistemas de colas de mensajes

Si consideramos lo que hemos dicho sobre los sistemas de colas de mensajes, parecería que han existido desde hace mucho en forma de implementaciones para servicios de correo electrónico. Los sistemas de correo electrónico se implementan, por lo general, a través de una colección de servidores de correo que almacenan y reenvían mensajes en beneficio de los usuarios de los sistemas directamente conectados al servidor. El enrutamiento a menudo se deja de lado, ya que los sistemas de correo electrónico pueden utilizar directamente los servicios de transporte subyacentes. Por ejemplo, en el protocolo de correo para internet SMTP (Postel, 1982), un mensaje se transfiere mediante la configuración de una conexión directa TCP hacia el servidor de correo de destino.

Lo que hace especiales a los sistemas de correo electrónico, comparados con los sistemas de colas de mensajes, es que se utilizan básicamente para proporcionar soporte directo a usuarios finales. Esto explica, por ejemplo, por qué un número de aplicaciones groupware se basa directamente en un sistema de correo electrónico (Khoshafian y Buckiewicz, 1995). Además, los sistemas de correo electrónico pueden tener requerimientos muy específicos, como el filtrado automático de mensajes, soporte para bases de datos de mensajería avanzada (por ejemplo, para recuperar mensajes almacenados previamente), etcétera.

Los sistemas generales de colas de mensajes no se utilizan sólo para soporte de usuarios finales. Una cuestión importante es que se configuran para habilitar la comunicación persistente entre procesos, sin importar si un proceso está ejecutando una aplicación de usuario, manejando el acceso a una base de datos, realizando cálculos, etc. Este método conduce a un conjunto de requerimientos diferente para los sistemas de colas de mensajes que para los sistemas de correo puros. Por ejemplo, los sistemas de correo electrónico generalmente no necesitan proporcionar garantías para

entrega de mensajes, prioridades de mensajes, facilidades de registro, multitransmisión eficiente, balanceo de cargas, tolerancia a fallas, y demás elementos de uso general.

Por tanto, los sistemas de colas de mensajes de propósito general tienen un amplio rango de aplicaciones, incluyendo correo electrónico, carga de trabajo, groupware, y procesamiento por lotes. Sin embargo, como ya establecimos, el área de aplicación más importante es la integración de una colección (posiblemente muy dispersa) de bases de datos y aplicaciones en un sistema de información federado (Hohpe y Woolf, 2004). Por ejemplo, una consulta que se expande a diversas bases de datos podría necesitar dividirse en subconsultas que se reenvíen a bases de datos individuales. Los sistemas de colas de mensajes ayudan a proporcionar los medios básicos para empacar cada subconsulta en un mensaje y enrutarlo hacia la base de datos adecuada. Otras facilidades de comunicación explicadas en este capítulo hasta el momento son más o menos adecuadas.

4.3.3 Ejemplo: sistema de colas de mensajes WebSphere de IBM

Para ayudarnos a comprender cómo funcionan en la práctica los sistemas de colas de mensajes, veamos un sistema específico, a saber, el sistema de cola de mensajes que es parte del producto WebSphere de IBM. Antes se le conocía como MQSeries, y ahora como **WebSphere MQ**. Hay mucha documentación sobre WebSphere MQ, y en adelante sólo podemos recurrir a los principios básicos. Muchos detalles arquitectónicos concernientes a redes de colas de mensajes pueden encontrarse en IBM (2005b, 2005d). Programar redes de colas de mensajes no es algo que pueda aprenderse en un domingo por la tarde, y la guía de programación de MQ (IBM, 2005a) es un buen ejemplo que muestra que ir de los principios a la práctica puede requerir de un gran esfuerzo.

Visión general

La arquitectura básica de una red de colas MQ es muy directa, según muestra la figura 4-22. Todas las colas son manejadas por **administradores de colas**. Un administrador de colas es responsable de eliminar mensajes de sus colas de envío, y de reenviarlos a otros administradores de colas. De igual manera, un administrador de colas es responsable de manejar mensajes de entrada; los toma de la red subyacente y luego almacena cada mensaje en la cola de entrada adecuada. Para tener una idea de lo que la mensajería puede significar: un mensaje tiene un tamaño máximo predeterminado de 4 MB, pero éste puede aumentar hasta 100 MB. Por lo general, una cola está restringida a 2 GB de información, pero según el sistema operativo subyacente, este máximo puede configurarse fácilmente para que sea mayor.

Los administradores de colas están conectados por pares a través de **canales de mensajes**, los cuales son una abstracción de las conexiones al nivel de transporte. Un canal de mensajes es una conexión unidireccional confiable, entre un administrador de cola de envío y uno de recepción, a través de la cual se transportan los mensajes en cola. Por ejemplo, un canal de mensajes basado en internet se implementa como una conexión TCP. Cada uno de los dos extremos de un canal de mensajes es administrado por un **agente de canal de mensajes** (**MCA**, por sus siglas en inglés). Un

MCA de envío prácticamente nada más verifica las colas de envío para un mensaje, lo envuelve en un paquete al nivel de transporte, y lo envía a través de la conexión con su MCA de recepción asociado. De igual manera, la tarea básica de un MCA de recepción es escuchar un paquete entrante, desenvolverlo, y posteriormente almacenarlo en la cola adecuada.

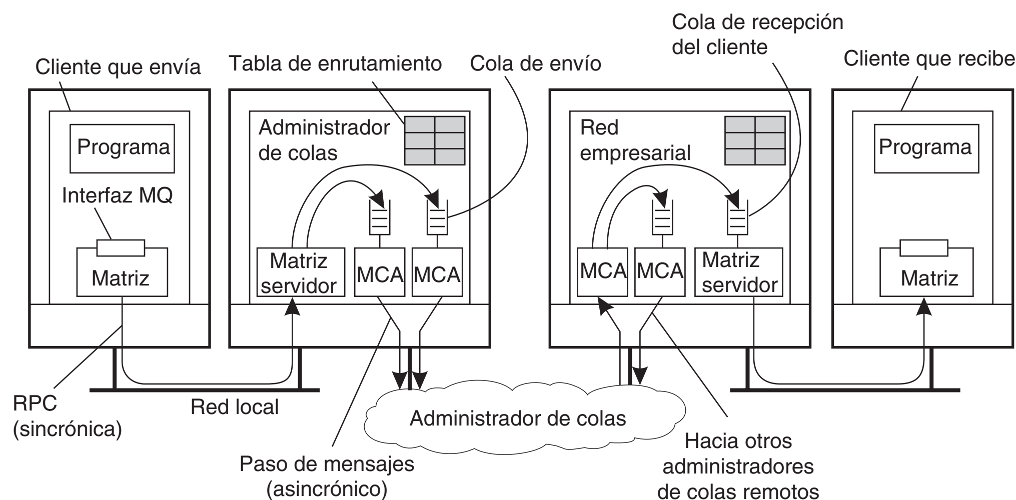


Figura 4-22. Organización general del sistema de cola de mensajes de IBM.

Los administradores de colas pueden vincularse al mismo proceso que la aplicación para la que administran las colas. En ese caso, las colas se ocultan de la aplicación detrás de una interfaz estándar, pero pueden ser manipuladas directamente por la aplicación. Una organización alterna es aquella donde los administradores de colas y las aplicaciones se ejecutan en máquinas separadas. En ese caso, a la aplicación se le ofrece la misma interfaz que cuando el administrador de colas se coloca en la misma máquina. Sin embargo, la interfaz se implementa como un proxy que se comunica con el administrador de colas mediante comunicación sincrónica basada en RCP tradicional. De esta manera, MQ básicamente conserva el modelo al que sólo pueden acceder las colas locales de una aplicación.

Canales

Un componente importante de MQ está formado por los canales de mensajes. Cada canal de mensaje tiene una cola de envío asociada desde la cual atrae los mensajes que debe transferir hacia el otro extremo. A lo largo del canal, la transferencia puede ocurrir sólo si sus MCA de envío y recepción están arriba y en ejecución. Además de poder iniciar ambos MCA manualmente, existen diversas formas alternas para iniciar un canal, algunas de las cuales explicaremos a continuación.

Una alternativa es hacer que una aplicación inicie directamente un extremo del canal, activando el MCA de envío o recepción. Sin embargo, desde un punto de vista de transparencia, ésta no es una alternativa atractiva. Un mejor método para iniciar un MCA de *envío* es configurar la cola de envío del canal para iniciar un disparador cuando un mensaje se coloque primero en la cola. Ese disparador está asociado con un manipulador para iniciar el MCA de envío en tal forma que pueda eliminar mensajes de la cola de envío.

Otra alternativa es iniciar un MCA sobre la red. En particular, si un lado de un canal ya está activo, el canal puede enviar un mensaje de control para solicitar se inicie el otro MCA. Tal mensaje de control se envía a un demonio que espera una dirección muy conocida en la misma máquina donde iniciará el otro MCA.

Los canales se detienen automáticamente después de pasado cierto tiempo durante el cual ningún otro mensaje llegó a la cola de envío.

Cada MCA tiene un conjunto de atributos asociados que determinan el comportamiento de todo el canal. Algunos de los atributos se encuentran en la figura 4-23. Los valores de los atributos de los MCA de envío y recepción deben ser compatibles, y tal vez negociados, antes de que pueda configurarse un canal. Por ejemplo, resulta evidente que ambos MCA deben soportar el mismo protocolo de transporte. Un ejemplo de un atributo no negociable, es si los mensajes se entregarán o no en el mismo orden en el que se colocan en la cola de envío. Si uno de los MCA quiere una entrega FIFO (PEPS), el orden debe respetarse. Un ejemplo de un valor de atributo negociable es la longitud máxima de mensaje, la cual se elegirá como el valor mínimo especificado por cualquiera de los MCA.

Atributo	Descripción
Tipo de transporte	Determina el protocolo de transporte a utilizarse
Entrega FIFO (PEPS)	Indica que los mensajes serán entregados en el orden en que se envíen
Longitud de mensaje	Longitud máxima de un solo mensaje
Cuenta de reintentos de configuración	Especifica el número máximo de reintentos permitidos para configurar un MCA remoto
Reintentos de entrega	Número máximo de veces que un MCA intentará colocar los mensajes recibidos en la cola

Figura 4-23. Algunos atributos asociados con agentes del canal de mensajes.

Transferencia de mensajes

Para transferir un mensaje desde un administrador de cola hacia otro (probablemente remoto), es necesario que cada mensaje lleve su dirección de destino, para lo cual se utiliza un encabezado de transmisión. Una dirección en MQ consta de dos partes: la primera parte contiene el nombre del administrador de cola al que se entregará el mensaje; y la segunda parte es el nombre de la cola de destino a la que el administrador adjuntará el mensaje.

Además de la dirección de destino, también es necesario especificar la ruta que debe seguir un mensaje. La especificación de la ruta se hace proporcionando el nombre de la cola local de envío a la que se adjuntará un mensaje. De esta manera, no es necesario proporcionar la ruta completa en un mensaje. Recuerde que cada canal de mensaje tiene exactamente una cola de envío. Al indicar a qué cola de envío se adjuntará un mensaje, especificamos efectivamente a qué administrador de cola se reenviará el mensaje.

En la mayoría de los casos, las rutas son almacenadas explícitamente dentro de un administrador de cola en una tabla de enrutamiento. Una entrada en una tabla de enrutamiento es un par (*destQM*, *sendQ*), donde *destQM* es el nombre del administrador de cola de destino, y *sendQ* es el nombre de la cola local de envío a la que debe adjuntarse el mensaje para ese administrador de cola. (En MQ, una entrada de la tabla de enrutamiento se conoce como alias.)

Es posible que sea necesario transferir un mensaje a través de varios administradores de cola antes de que llegue a su destino. Siempre que un administrador de cola intermediario reciba el mensaje, simplemente extraerá el nombre del administrador de cola de destino del encabezado del mensaje, y hará una búsqueda en la tabla de enrutamiento para encontrar la cola local de envío a la que el mensaje debe adjuntarse.

Es importante observar que cada administrador de cola tiene un nombre único en el sistema que efectivamente se usa como identificador para ese administrador de cola. El problema con el uso de estos nombres, es que al remplazar a un administrador de cola, o al cambiar su nombre, se afectarán todas las aplicaciones que le envían mensajes. Estos problemas pueden aligerarse con el uso de un **alias local** para nombres de administradores de cola. Un alias definido dentro de un administrador de cola *M1* es otro nombre para un administrador de cola *M2*, pero que está disponible sólo para aplicaciones de interfaz con *M1*. Un alias permite el uso del mismo nombre (lógico) de una cola, incluso si cambia el administrador de cola de esa cola. Cambiar el nombre de un administrador de cola requiere que cambiemos su alias en todos los administradores de cola. Sin embargo, las aplicaciones no se verán afectadas.

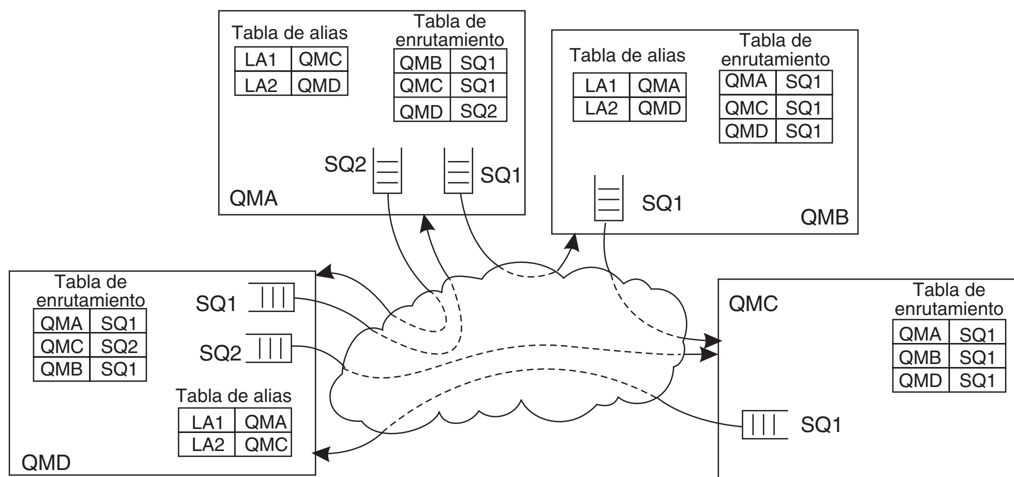


Figura 4-24. Organización general de una red de colas MQ que utiliza tablas de enrutamiento y de alias.

El principio para utilizar tablas de enrutamiento y alias aparece en la figura 4-24. Por ejemplo, una aplicación vinculada al administrador de cola *QMA* puede referirse a un administrador remoto de cola mediante el alias local *LA1*. El administrador de cola primero buscará el destino real en la tabla de alias para descubrir que es un administrador de cola *QMC*. La ruta para *QMC* está en la tabla de enrutamiento, la cual establece que los mensajes para *QMC* deben adjuntarse a la cola de salida *SQ1*, la cual se utiliza para transferir mensajes al administrador de cola *QMB*. Este último utilizará su tabla de enrutamiento para reenviar el mensaje a *QMC*.

Al seguir este método de enrutamiento y uso de alias se llega a una interfaz de programación que es, por principio, relativamente simple, conocida como **Interfaz de Cola de Mensajes** (**MQI**, por sus siglas en inglés). Las primitivas más importantes de la MQI aparecen en la figura 4-25.

Primitiva	Descripción
MQopen	Abre una cola (probablemente remota)
MQclose	Cierra una cola
MQput	Coloca un mensaje en una cola abierta
MQget	Obtiene un mensaje de una cola (local)

Figura 4-25. Primitivas disponibles en la interfaz de cola de mensajes.

Para colocar mensajes en una cola, una aplicación llama a la primitiva MQopen, especificando una cola de destino en un administrador específico de cola. El administrador de cola puede nombrarse mediante el uso de un alias local disponible. Ya sea que la cola de destino sea remota o no, es completamente transparente para la aplicación. También es necesario llamar a MQopen si la aplicación quiere obtener mensajes a partir de su cola local. Solamente las colas locales pueden abrirse para leer mensajes entrantes. Cuando una aplicación termina su acceso a una cola, ésta debe cerrar mediante una llamada a MQclose.

Los mensajes pueden escribirse para una cola, o leerse desde una cola, mediante MQput y MQget, respectivamente. En principio, los mensajes se eliminan de una cola de acuerdo con su prioridad. Los mensajes con la misma prioridad se eliminan conforme al principio de primero en entrar, primero en salir; es decir, el mensaje pendiente de más larga permanencia se elimina primero. También es posible solicitar mensajes específicos. Por último, MQ proporciona herramientas para indicar a las aplicaciones que ha llegado un mensaje, y así evitar que una aplicación tenga que preguntar continuamente por mensajes entrantes a la cola.

Administración de redes sobrepuestas

Por la descripción que hemos hecho hasta el momento, debe resultar claro que una parte importante de la administración de sistemas MQ es conectar a los diversos administradores de colas en una red sobrepuesta consistente. Más aún, con el tiempo, esta red necesita mantenimiento. Para redes pequeñas, este mantenimiento no requerirá más que un trabajo administrativo promedio, pero las

cosas se complican cuando la cola de mensajes se utiliza para integrar y desintegrar grandes sistemas existentes.

Un asunto más importante es que las redes sobrepuestas deben administrarse manualmente. Esta administración no sólo involucra la creación de canales entre administradores de cola, sino también el llenado de las tablas de enrutamiento. Evidentemente, esto puede crecer hasta convertirse en una pesadilla. Por desgracia, el soporte para administración de sistemas MQ es avanzado sólo en el sentido de que un administrador puede configurar virtualmente cualquier posible atributo, y ajustar cualquier configuración concebible. Sin embargo, el resultado final es que a los canales y a las tablas de enrutamiento se les debe dar mantenimiento manualmente.

En el núcleo de la administración sobrepuesta se encuentra la **función de control de canales**, la cual lógicamente se ubica entre los agentes del canal de mensajes. Este componente permite que un operador dé un seguimiento exacto a lo que sucede en dos puntos finales de un canal. Además, se utiliza para crear canales y tablas de enrutamiento, pero también para manejar a los administradores de cola que hospedan a los agentes del canal de mensajes. De cierto modo, este método para administración de sobrepuestas se parece bastante a la administración de servidores cluster donde se utiliza un solo servidor de administración. En el último caso, el servidor ofrece esencialmente sólo un shell remoto a cada máquina del cluster, junto con algunas operaciones colectivas para manejar grupos de máquinas. La buena noticia sobre la administración de sistemas distribuidos es que ofrece muchas oportunidades si usted está buscando un área para explorar nuevas soluciones a problemas serios.

4.4 COMUNICACIÓN ORIENTADA A FLUJOS

La comunicación, como la hemos explicado hasta el momento, se ha concentrado en el intercambio más o menos independiente y completo de unidades de información. Algunos ejemplos incluyen una petición para invocar un procedimiento, la respuesta a tal petición, y mensajes intercambiados entre aplicaciones como en los sistemas de colas de mensajes. La principal característica de este tipo de comunicación es que no importa en qué punto del tiempo en particular ocurre la comunicación. Aunque un sistema puede comportarse de forma muy lenta o muy rápidamente, la sincronización no tiene efecto alguno sobre la integridad.

También hay formas de comunicación en las que la sincronización juega un papel crucial. Por ejemplo, considere un flujo de audio concebido como una secuencia de muestras de 16 bits, donde cada muestra representa la amplitud de la onda sonora como se hace con la Modulación de Código de Pulso (PCM, por sus siglas en inglés). También suponga que el flujo de audio representa calidad de disco compacto, lo cual significa que la onda sonora original se ha muestreado a una frecuencia de 44100 Hz. Para reproducir el sonido original, es esencial que las muestras del flujo de audio se reproduzcan en el orden en que aparecen en el flujo, pero también en intervalos de exactamente 1/44100 segundos. La reproducción a una velocidad diferente producirá una versión incorrecta del sonido original.

La pregunta que nos hacemos en esta sección es qué herramientas debe ofrecer un sistema distribuido para intercambiar información dependiente del tiempo tal como los flujos de audio y video.

En Halsall (2001) se explican diversos protocolos de red que tratan con la comunicación orientada a flujos. Steinmetz y Nahrstedt (2004) proporcionan una introducción general a cuestiones multimedia, parte de la cual forma la comunicación orientada a flujos. Babcock y colaboradores (2002) explican el procesamiento de consultas en flujos de datos.

4.4.1 Soporte para medios continuos

El soporte para intercambiar información dependiente del tiempo con frecuencia se conoce como soporte para medios continuos. Un medio se refiere al recurso mediante el cual se transmite la información. Estos recursos incluyen a los medios de almacenamiento y transmisión, medios de presentación como un monitor, etc. Un tipo importante de medio es la forma en que se *representa* la información. En otras palabras, ¿cómo se codifica la información en un sistema de cómputo? Se utilizan diferentes representaciones para diferentes tipos de información. Por ejemplo, el texto generalmente se codifica en ASCII o Unicode. Las imágenes pueden representarse en diferentes formatos, como GIF o JPEG. En un sistema de cómputo, los flujos de audio pueden codificarse al tomar muestras de 16 bits empleando PCM.

En **medios continuos**, las relaciones temporales entre diferentes elementos de datos resultan fundamentales para interpretar correctamente lo que significan en realidad los datos. Ya dimos como ejemplo, la reproducción de una onda sonora mediante la reproducción de un flujo de audio. Como otro ejemplo, consideremos el movimiento. El movimiento puede representarse mediante una serie de imágenes en la que deben desplegarse imágenes sucesivas en un espacio T uniforme en el tiempo, por lo general de 30 a 40 milisegundos por imagen. Una reproducción correcta no sólo requiere mostrar las imágenes en el orden apropiado, sino también a una frecuencia constante de $1/T$ imágenes por segundo.

Por contraste con los medios continuos, los **medios discretos** se caracterizan por el hecho de que las relaciones temporales entre elementos de datos *no* son fundamentales para interpretar correctamente los datos. Ejemplos típicos de medios discretos incluyen las representaciones de texto y fotogramas, pero también código objeto o archivos ejecutables.

Flujo de datos

Para captar el intercambio de información dependiente del tiempo, los sistemas distribuidos generalmente proporcionan soporte para **flujos de datos**. Un flujo de datos no es otra cosa más que una secuencia de unidades de datos. Estos flujos pueden aplicarse tanto a medios discretos como a medios continuos. Las tuberías en UNIX o las conexiones TCP/IP son ejemplos típicos de flujos discretos de datos (orientados a bytes). Para reproducir un archivo de audio se necesita, por lo general, configurar un flujo continuo de datos entre el archivo y el dispositivo de audio.

La sincronización resulta crucial para los flujos continuos de datos. Para captar los aspectos de la sincronización, a menudo se marca la diferencia entre los diferentes modos de transmisión. En el **modo de transmisión asíncrona**, los elementos de datos de un flujo se transmiten uno después de otro, pero no hay más restricciones de sincronización en cuanto a cuándo debe ocurrir la transmi-

sión de elementos. Éste generalmente es el caso para los flujos discretos de datos. Por ejemplo, un archivo puede transferirse como un flujo de datos, pero es prácticamente irrelevante cuándo exactamente se completa la transferencia de cada elemento.

En el **modo de transmisión síncrono**, existe un retraso máximo fin a fin para cada unidad del flujo de datos. Si una unidad de datos se transfiere mucho más rápido que el retraso máximo tolerado no es importante. Por ejemplo, un sensor puede muestrear la temperatura a cierta velocidad y pasarla al operador a través de una red. En ese caso, puede ser importante garantizar que el tiempo de propagación fin a fin a través de la red sea menor que el intervalo de tiempo transcurrido entre la toma de muestras, pero no resulta dañino si las muestras se propagan mucho más rápido de lo necesario.

Por último, en el **modo de transmisión isócrono** es necesario que las unidades de datos se transfieran a tiempo. Esto significa que la transferencia de datos está sujeta a un retraso máximo *y* mínimo fin a fin, también conocido como inestabilidad limitada (retraso). El modo de transmisión isócrono resulta particularmente interesante para sistemas multimedia distribuidos, ya que desempeña un papel muy importante en la representación de audio y video. En este capítulo, consideraremos sólo flujos continuos de datos con el uso de transmisión isócrona, a la cual nos referiremos simplemente como flujos.

Los flujos pueden ser simples o complejos. Un **flujo simple** consiste únicamente en una sola secuencia de datos, mientras que un **flujo complejo** consta de varios flujos simples relacionados, llamados **subflujos**. La relación entre los subflujos de un flujo complejo a menudo también depende del tiempo. Por ejemplo, el audio estereofónico puede transmitirse por medio de un complejo flujo constituido por dos subflujos, donde cada flujo es utilizado para ocupar un solo canal de audio. Sin embargo, es importante que esos dos subflujos estén continuamente sincronizados. En otras palabras, las unidades de datos de cada flujo se comunicarán por pares para garantizar el efecto estereofónico. Otro ejemplo de un flujo complejo es el que se utiliza para transmitir una película. Tal flujo podría consistir en un solo flujo de video junto con dos flujos para la transmisión del sonido estereofónico de la película. Un cuarto flujo podría contener subtítulos para sordos o una traducción a un lenguaje diferente al del audio. De nuevo, la sincronización de los subflujos es importante. Si la sincronización falla, la reproducción de la película también fallará. Más adelante retomaremos el tema de la sincronización de flujos.

Desde la perspectiva de los sistemas distribuidos, podemos distinguir diversos elementos necesarios para soportar los flujos. Por simplicidad, nos concentraremos en el flujo de datos almacenados, lo contrario al flujo de datos en vivo. En el último caso, los datos se capturan en tiempo real y se envían por la red hacia los destinatarios. La principal diferencia entre estos dos tipos de datos es que el flujo de datos en vivo brinda menos oportunidades de ajuste. De acuerdo con Wu y colaboradores (2001), podemos esquematizar una arquitectura general cliente-servidor para dar soporte a flujos continuos multimedia tal como indica la figura 4-26.

Esta arquitectura general revela una gran cantidad de asuntos que se deben tratar. En primer lugar, los datos multimedia, en mayor medida el video y en menor medida el audio, deberán comprimirse de manera importante para reducir el almacenamiento requerido, y especialmente la capacidad de la red. Desde una perspectiva de comunicación, resulta más importante controlar la calidad de transmisión y la sincronización. Enseguida explicaremos estos temas.

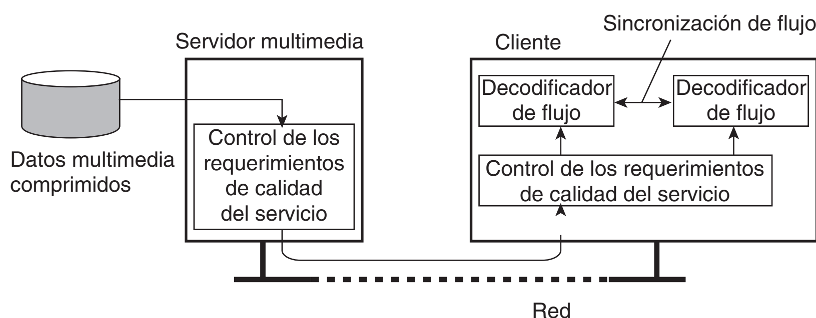


Figura 4-26. Arquitectura general para pasar a través de una red un flujo de datos multimedia almacenados.

4.4.2 Flujos y calidad del servicio

Los requerimientos de sincronización (y otros no funcionales) se expresan generalmente como requerimientos de **Calidad del Servicio (QoS**, por sus siglas en inglés). Estos requerimientos describen lo que se necesita del sistema distribuido subyacente y de la red para garantizar que, por ejemplo, las relaciones temporales de un flujo puedan preservarse. La QoS para flujos continuos de datos tiene que ver principalmente con puntualidad, volumen, y confiabilidad. En esta sección veremos la QoS y su relación con la configuración de un flujo.

Mucho se ha dicho sobre cómo especificar la QoS requerida (por ejemplo, consulte a Jin y Nahrstedt, 2004). Desde la perspectiva de una aplicación, en muchos casos esto se reduce a especificar algunas propiedades importantes (Halsall, 2001):

1. La velocidad de bits requerida a la que deben transportarse los datos.
2. El retraso máximo hasta que se haya configurado una sesión (es decir, cuando una aplicación puede comenzar el envío de datos).
3. El retraso máximo fin a fin (es decir, cuánto tiempo le llevará a una unidad de datos llegar hasta un destinatario).
4. La varianza del retraso máximo, o inestabilidad.
5. El retraso máximo de un ciclo.

Resulta importante advertir que es posible realizar muchas mejoras a estas especificaciones tal como, por ejemplo, explican Steinmetz y Nahrstadt (2004). Sin embargo, cuando se trata de la comunicación orientada a flujos que se basa en el protocolo de pila de internet, simplemente debemos vivir con el hecho de que la base de la comunicación está formada por un servicio extremadamente simple de datagramas IP. Cuando lo anterior es un hecho, como fácilmente puede ser el caso en internet, especificar la IP permite a la implementación de un protocolo deshacerse de paquetes cuando lo juzgue conveniente. Muchos de los sistemas distribuidos, si no es que todos, que soportan la

comunicación orientada a flujos se basan actualmente en el protocolo de pila de internet. Demasiado para especificaciones QoS. (En realidad, IP proporciona cierto soporte de QoS, pero rara vez se implementa.)

Cómo imponer la QoS

Dado que el sistema subyacente sólo ofrece un servicio bien intencionado de entrega, un sistema distribuido puede intentar ocultar lo más posible la *carencia* de calidad del servicio. Por fortuna, existen diversos mecanismos que se pueden utilizar.

Primero, la situación no es realmente tan mala como se planteó hasta el momento. Por ejemplo, internet proporciona recursos para diferenciar las clases de datos mediante sus **servicios diferenciados**. Un servidor de envío puede marcar paquetes de salida como pertenecientes a una o a varias clases, incluyendo la clase de **reenvío expedito** que esencialmente especifica que un paquete debe reenviarse mediante el ruteador actual con absoluta prioridad (Davie y cols., 2002). Además, también hay una clase de **reenvío garantizado**, por el cual el tráfico se divide en cuatro subclases, junto con tres formas para eliminar paquetes si la red se congestiona. Por tanto, el reenvío garantizado define de manera efectiva un rango de prioridades que pueden asignarse a paquetes, y como tal permite a las aplicaciones diferenciar paquetes sensibles al tiempo de aquellos que no son críticos.

Además de estas soluciones al nivel de red, un sistema distribuido también puede ayudar a que los destinatarios envíen información. Aunque generalmente no hay tantas herramientas disponibles, una que es particularmente útil es el uso de un bufer para reducir la inestabilidad. El principio es sencillo, como indica la figura 4-27. Si suponemos que los paquetes se retrasan con cierta varianza cuando se transmiten por la red, el destinatario simplemente los almacena en un bufer durante un tiempo máximo. Esto permitirá al destinatario pasar paquetes a la aplicación con regular velocidad con la certeza de que siempre habrá suficientes paquetes entrando al bufer para su reproducción a esa velocidad.

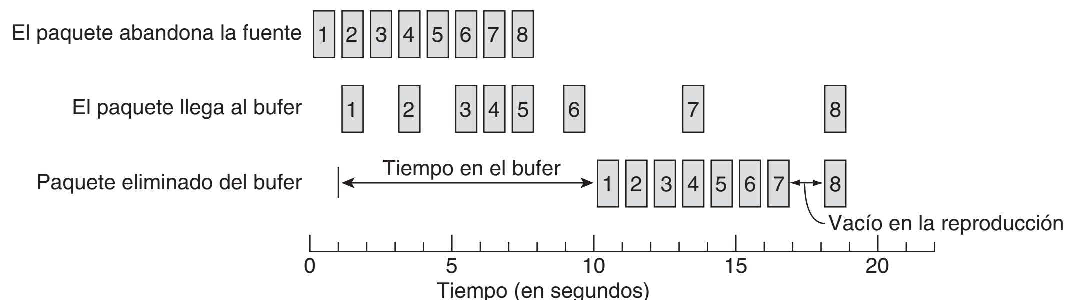


Figura 4-27. Uso de un bufer para reducir la inestabilidad.

Por supuesto, las cosas pueden empeorar, como ilustra el paquete #8 de la figura 4-27. El tamaño del bufer destinatario corresponde a 9 segundos de paso de paquetes a la aplicación. Desgraciadamente, al paquete #8 le lleva 11 segundos alcanzar al destinatario, momento en el que el bufer

se habrá vaciado por completo. El resultado es un vacío en la reproducción de la aplicación. La única solución es incrementar el tamaño del bufer. La desventaja evidente es que el retraso en que la aplicación destinataria puede iniciar la reproducción de los datos contenidos en los paquetes también se incrementa.

También es posible utilizar otras técnicas. Darnos cuenta de que estamos tratando con un servicio subyacente bien intencionado también significa que los paquetes pueden perderse. Para compensar esta pérdida en la calidad del servicio necesitamos aplicar técnicas de corrección de errores (Perkins y cols., 1998; y Wah y cols., 2000). Solicitar al remitente que retransmita un paquete, por lo general, está fuera de lugar, así que debe aplicarse la **corrección de errores de reenvío** (**FEC**, por sus siglas en inglés). Una técnica muy conocida es codificar los paquetes de salida en tal forma que cualquier salida *k* de *n* paquetes recibidos sea suficiente para reconstruir *k* paquetes correctos.

Un problema que puede presentarse es que un solo paquete contenga diversas tramas de audio y video. En consecuencia, cuando se pierde un paquete, el destinatario puede percibir un gran vacío cuando reproduzca las tramas. Es posible sortear este efecto interpolando las tramas, como ilustra la figura 4-28. De esta manera, cuando se pierde un paquete, el vacío resultante en tramas posteriores se distribuye en el tiempo. Sin embargo, observe que este método requiere de un bufer destinatario más grande, en comparación con la no interpolación, y por tanto impone un retraso de inicio más alto para la aplicación destinataria. Por ejemplo, cuando consideramos la figura 4-28(b), para reproducir las primeras cuatro tramas el destinatario necesitará haber entregado cuatro paquetes, en lugar de sólo uno, en comparación con la transmisión no interpolada.

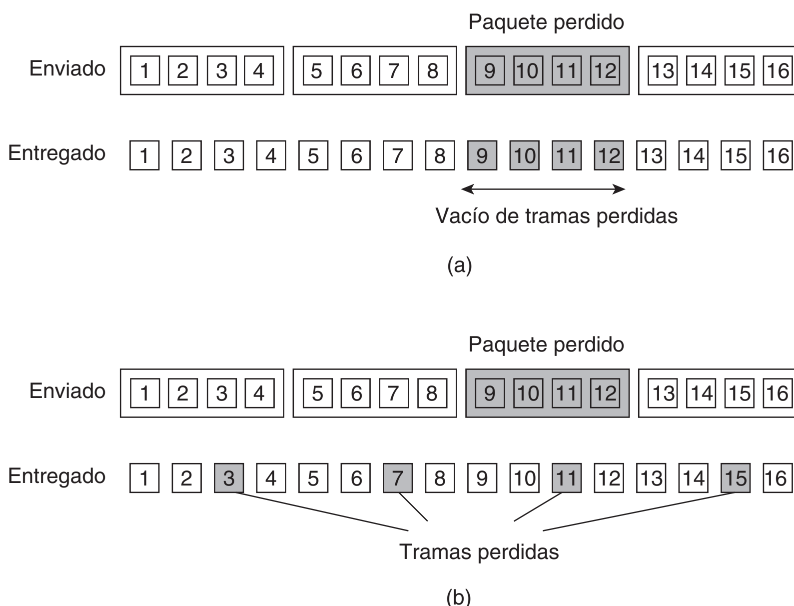


Figura 4-28. Efecto de un paquete perdido en (a) una transmisión no interpolada, y en (b) una transmisión interpolada.

4.4.3 Sincronización de flujos

Un tema muy importante en sistemas multimedia es que flujos diferentes, posiblemente en forma de un flujo complejo, están mutuamente sincronizados. La sincronización de flujos tiene que ver con mantener las relaciones temporales entre flujos. Ocurren dos tipos de sincronización.

La forma de sincronización más sencilla sucede entre un flujo discreto de datos y un flujo continuo de datos. Por ejemplo, consideremos una diapositiva mejorada con audio y mostrada en la web. La diapositiva se transfiere del servidor al cliente en forma de un flujo de datos discreto. Al mismo tiempo, el cliente debe reproducir una parte específica de un flujo de audio que coincida con la diapositiva en curso y también sea traído desde el servidor. En este caso, el flujo de audio se sincroniza con la presentación de diapositivas.

Un tipo más demandante de sincronización es el que tiene lugar entre flujos continuos de datos. Un ejemplo diario es la reproducción de una película en la que el flujo de video necesita sincronizarse con el de audio, proceso conocido como sincronización de labios. Otro ejemplo de sincronización es la reproducción de un flujo de audio estereofónico que consiste en dos subflujos, uno para cada canal. Una reproducción adecuada requiere que los dos subflujos estén muy bien sincronizados: una diferencia de más de 20 ms puede distorsionar el efecto estereofónico.

La sincronización ocurre al nivel de las unidades de datos que conforman el flujo. En otras palabras, podemos sincronizar dos flujos sólo entre unidades de datos. La elección de la unidad de datos depende en gran medida del nivel de abstracción con que se considere al flujo de datos. Para concretar, consideremos nuevamente un flujo de audio (de un solo canal) de la calidad de un disco compacto. Con la granulación más fina, tal flujo aparece como una secuencia de muestras de 16 bits. Con una frecuencia de muestreo de 44 100 Hz, la sincronización con otros flujos de audio podría, en teoría, ocurrir aproximadamente cada 23 µs. Para efectos estereofónicos de más alta calidad, es evidente que se necesita la sincronización a este nivel.

Sin embargo, cuando consideramos la sincronización entre un flujo de audio y uno de video para sincronización de labios, podemos emplear una granulación más gruesa. Como explicamos, las tramas de video deben desplegarse a una velocidad de 25 Hz o más. Si consideramos el ampliamente utilizado estándar NTSC de 29.97 Hz, podremos agrupar muestras de audio en unidades lógicas que permanecen mientras dura la reproducción de una trama de video (33 ms). Con una frecuencia de muestreo de audio de 44 100 Hz, una unidad de datos de audio puede entonces ser tan grande como 1 470 muestras u 11 760 bytes (si suponemos que cada muestra es de 16 bits). En la práctica, pueden tolerarse unidades más grandes que duran hasta 40 o incluso 80 ms (Steinmetz, 1996).

Mecanismos de sincronización

Ahora veamos cómo se lleva a cabo la sincronización. Debemos diferenciar dos puntos: (1) los mecanismos básicos para sincronizar dos flujos, y (2) la distribución de esos mecanismos en un ambiente de red.

Los mecanismos de sincronización pueden considerarse desde varios niveles de abstracción diferentes. En el nivel más bajo, la sincronización se realiza explícitamente operando las unidades

de datos de flujos simples. Este principio aparece en la figura 4-29. En esencia, hay un proceso que simplemente ejecuta operaciones de lectura y escritura en varios flujos simples, garantizando que tales operaciones se apegan a restricciones de sincronización específicas.

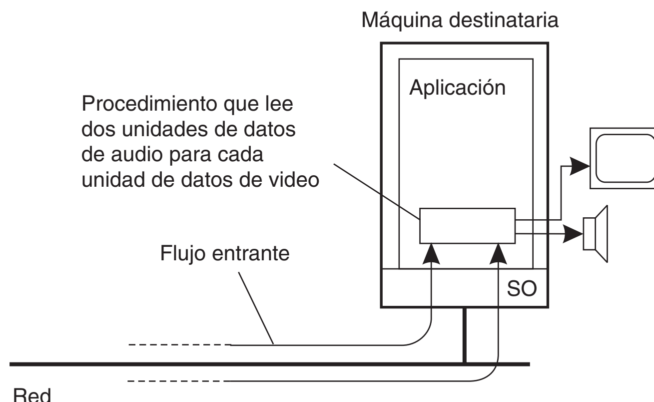


Figura 4-29. Principio de sincronización explícita al nivel de unidades de datos.

Por ejemplo, consideremos una película que se presenta como dos flujos de entrada. El flujo de video contiene imágenes de baja calidad descomprimidas de 320 × 240 píxeles, cada píxel está codificado mediante un solo byte, lo cual produce unidades de datos de video de 76 800 bytes por unidad. Suponga que las imágenes van a desplegarse a 30 Hz, o una imagen cada 33 ms. Se supone que el flujo de audio contiene muestras agrupadas en unidades de 11 760 bytes, y que cada unidad corresponde a 33 ms de audio, como explicamos antes. Si el proceso de entrada puede manejar 2.5 MB/s, podemos lograr la sincronización de labios si alternamos simplemente cada 33 ms la lectura de una imagen y la lectura de un bloque de muestras de audio.

La desventaja de este método es que la aplicación se hace totalmente responsable de implementar la sincronización cuando sólo tiene a su disposición herramientas de bajo nivel. Un mejor método es ofrecerle a una aplicación una interfaz que le permita controlar más fácilmente flujos y dispositivos. Volviendo a nuestro ejemplo, suponga que la reproducción del video tiene una interfaz de control que le permite especificar la velocidad a la cual deben aparecer las imágenes. Además, la interfaz ofrece la herramienta de registrar un controlador definido por el usuario que es llamado cada vez que llegan *k* nuevas imágenes. El dispositivo de audio ofrece una interfaz análoga. Con estas interfaces de control, un desarrollador de aplicaciones puede escribir un sencillo programa monitor constituido por dos controladores, uno para cada flujo, que verifiquen conjuntamente si el flujo de audio y el de video están lo suficientemente sincronizados, y si es necesario, que ajuste la velocidad a la cual se presentan las unidades de video o de audio.

Este último ejemplo se ilustra en la figura 4-30, y es típico para muchos sistemas middleware multimedia. En efecto, el middleware multimedia ofrece una colección de interfaces para controlar flujos de audio y video que incluyen interfaces para controlar dispositivos como monitores, cámaras,

micrófonos, etc. Cada dispositivo y cada flujo tienen sus propias interfaces de alto nivel, incluyendo interfaces para notificar a una aplicación cuándo ocurre algún evento. Las últimas interfaces se utilizan posteriormente para escribir controladores para sincronización de flujos. Blair y Stefani (1998) proporcionan ejemplos de tales interfaces.

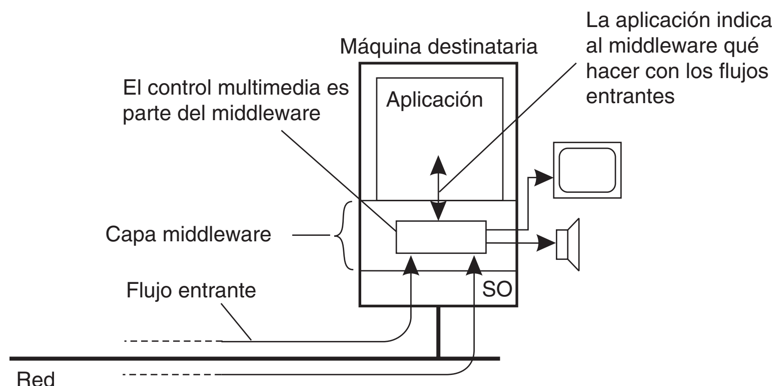


Figura 4-30. Principio de sincronización tal como la soportan las interfaces de alto nivel.

La distribución de mecanismos de sincronización es otro tema que debemos revisar. Primero, el lado destinatario de un flujo complejo consistente en subflujos que requieren sincronización necesita saber exactamente qué hacer. En otras palabras, debe tener una *especificación de sincronización* completa disponible localmente. Una práctica común es proporcionar implícitamente esta información mediante la multiplexación de los diferentes flujos en un solo flujo que contenga todas la unidades de datos, incluso aquellas necesarias para la sincronización.

El último método para sincronizar es seguido por flujos MPEG. Los estándares **MPEG** (**Motion Picture Experts Group**) forman una colección de algoritmos para comprimir video y audio. Existen diversos estándares MPEG. Por ejemplo, MPEG-2 fue diseñado originalmente para comprimir video de calidad de difusión de 4 a 6 Mbps. En MPEG-2, un número ilimitado de flujos discretos y continuos pueden combinarse en un solo flujo. Cada flujo de entrada se convierte primero en un flujo de paquetes que lleva un registro de tiempo basado en un reloj de sistema de 90 kHz. Estos flujos son posteriormente multiplexados en un **flujo de programa** que entonces consiste en paquetes de longitud variable, pero los cuales tienen en común que todos poseen la misma base de tiempo. El lado destinatario desmultiplexa el flujo, utilizando nuevamente los registros de tiempo de cada paquete como el mecanismo básico para efectuar la sincronización de interflujo.

Otro asunto importante es si la sincronización debe ocurrir del lado remitente o del lado destinatario. Si el remitente maneja la sincronización, es posible combinar los flujos en uno solo con un tipo de unidad de datos diferente. De nuevo, consideremos un flujo de audio estéreo que consiste en dos subflujos, uno para cada canal. Una posibilidad es transferir cada flujo independientemente del destinatario y dejar que el último sincronice las muestras a manera de

pares. Desde luego, debido a que cada subflujo puede estar sujeto a un retraso diferente, la sincronización puede resultar extremadamente difícil. Un mejor método es combinar los dos subflujos en el remitente. El flujo resultante consiste en unidades de datos que constan de pares de muestras, una para cada canal. El destinatario ahora simplemente tiene que leer cada unidad de datos y dividirla en una muestra izquierda y otra muestra derecha. Los retrasos para ambos canales son ahora idénticos.

4.5 COMUNICACIÓN POR MULTITRANSMISIÓN

Un tema importante en comunicación de sistemas distribuidos es el soporte para enviar datos a varios destinatarios, lo cual también se conoce como comunicación por multitransmisión. Por muchos años, este aspecto ha pertenecido al dominio de los protocolos de red, donde se han implementado y evaluado diversas propuestas de solución al nivel de red y al nivel de transporte (Janic, 2005; y Obraczka, 1998). Un asunto importante en todas las soluciones fue configurar rutas de comunicación para diseminar información. En la práctica, esto involucró un esfuerzo enorme de administración que en muchos casos requería la intervención humana. Además, como no hay propuestas que coincidan, los ISP se han mostrado reacios a soportar la multitransmisión (Diot y cols., 2000).

Con la llegada de la tecnología de punto a punto, y la notablemente estructurada administración sobrepuesta, se volvió más sencillo configurar rutas de comunicación. Debido a que las soluciones de punto a punto se utilizan típicamente en la capa de aplicación, se han introducido varias técnicas de multitransmisión al nivel de aplicación. En esta sección analizaremos brevemente estas técnicas.

La comunicación por multitransmisión también puede lograrse de maneras diferentes a la configuración explícita de rutas de comunicación. Como veremos también en esta sección, la diseminación de información basada en gossiping proporciona formas sencillas (aunque con frecuencia menos eficientes) de lograr la multitranmisión.

4.5.1 Multitransmisión al nivel de aplicación

La idea básica de la multitransmisión al nivel de aplicación es que los nodos se organizan en una red sobrepuesta, la cual después se utiliza para diseminar la información a sus miembros. Una observación importante es que los ruteadores de red no están organizados en grupos de miembros. En consecuencia, las conexiones entre los nodos de la red sobrepuesta pueden cruzar diversas uniones físicas, y como tal, los mensajes de enrutamiento dentro de la sobrepuesta pueden no ser óptimos en comparación con lo que se hubiese logrado mediante un enrutamiento al nivel de red.

Un asunto crucial de diseño es la construcción de la red sobrepuesta. En esencia, hay dos métodos (El-Sayed, 2003). Primero, los nodos pueden organizarse por sí mismos de manera directa en un árbol, lo cual significa que existe una ruta única (sobrepuesta) entre cada par de nodos. Un método alterno es que los nodos se organicen en una red acoplada en la que cada nodo tendrá varios

vecinos y, en general, existen múltiples rutas entre cada par de nodos. La principal diferencia entre los dos enfoques, es que generalmente el último proporciona mayor fuerza: si una conexión falla (digamos, porque un nodo falla), aún existirá una oportunidad de diseminar la información sin tener que reorganizar de inmediato toda la red sobrepuesta.

Para concretar, consideremos un esquema relativamente simple para la construcción de un árbol de multitransmisión en Chord, el cual explicamos en el capítulo 2. Este esquema se propuso originalmente para Scribe (Castro y cols., 2002), y es un esquema de multitransmisión al nivel de aplicación, basado en Pastry (Rowstron y Druschel, 2001). El último también es un sistema de punto a punto basado en DHT.

Suponga que un nodo quiere iniciar una sesión multitransmisión. Para lograrlo, simplemente genera un identificador de multitransmisión, digamos *mid*, el cual es tan sólo una llave de 160 bits elegida al azar. Después busca a *succ*(*mid*), el nodo responsable de esa llave, y promueve que se vuelva la raíz del árbol de multitransmisión que se utilizará para enviar información a los nodos interesados. Para unirse a un árbol, un nodo *P* simplemente ejecuta la operación LOOKUP(mid), la cual tiene el efecto de que el mensaje de búsqueda con la petición de unir al grupo por multitransmisión *mid* se enrutará desde *P* hasta *succ*(*mid*). Como ya mencionamos, en el capítulo 5 explicaremos el propio algoritmo de enrutamiento.

En su camino hacia la raíz, la petición de unión pasará a varios nodos. Suponga que primero alcanza al nodo *Q*. Si *Q* nunca ha visto una petición de unión para *mid*, se convertirá en un **promotor** para ese grupo. En ese punto, *P* se volverá un hijo de *Q*, mientras que *Q* seguirá reenviando la petición de unión hacia la raíz. Si el siguiente nodo de la raíz, digamos *R*, aún no es un promotor, se volverá uno y registrará a *Q* como su hijo, y seguirá enviando la petición de unión.

Por otra parte, si *Q* (o *R*) ya es un promotor de *mid*, también registrará al remitente anterior como su hijo (es decir, *P* o *Q*, respectivamente), pero ya no habrá necesidad de enviar una vez más la petición de unión a la raíz, debido a que *Q* (o *R*) ya será un miembro del árbol de multitransmisión.

Los nodos como *P* que han solicitado explícitamente la unión con el árbol de multitransmisión también son, por definición, promotores. El resultado de este esquema es que construimos un árbol de multitransmisión a través de la red sobrepuesta con dos tipos de nodos: promotores puros que actúan como ayudantes, y nodos que también son promotores pero que han solicitado explícitamente la unión con el árbol. La multitransmisión ahora es sencilla: un nodo simplemente envía un mensaje por multitransmisión hacia la raíz del árbol ejecutando nuevamente la operación LOOKUP(mid), después de lo cual ese mensaje puede enviarse a lo largo del árbol.

Observemos que esta descripción de alto nivel de la multitransmisión en Scribe no le hace justicia a su diseño original. Invitamos al lector interesado a que consulte los detalles, los cuales puede encontrar en Castro y colaboradores (2002).

Construcción sobrepuesta

A partir de la descripción de alto nivel proporcionada líneas antes, debe resultar claro que aunque construir un árbol no es tan difícil, una vez organizados los nodos en una sobrepuesta, construir un árbol eficiente puede ser una historia muy distinta. Advierta que en nuestra descripción la selección

de los nodos que participan en el árbol no toma en cuenta ninguna métrica de rendimiento: simplemente se basa en el enrutamiento (lógico) de mensajes a través de la sobrepuesta.

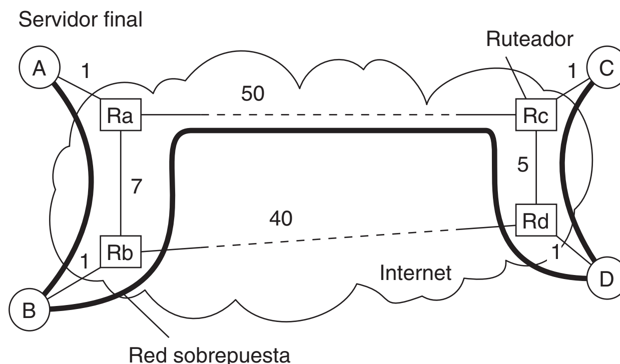


Figura 4-31. Relación entre vínculos de una sobrepuesta y las rutas reales al nivel de red.

Para comprender el problema actual, observe la figura 4-31 que muestra un pequeño conjunto de cuatro nodos organizados en una sencilla red sobrepuesta, en la que el nodo *A* forma la raíz de un árbol de multitransmisión. Los costos de atravesar un vínculo físico también aparecen. Ahora, siempre que *A* multidifunda un mensaje a los otros nodos, veremos que este mensaje atravesará dos veces cada uno de los vínculos <*B*, *Rb*>, <*Ra*, *Rb*>, <*Rc*, *Rd*> y <*D*, *Rd*>. La red sobrepuesta podría haber sido más eficiente si no hubiésemos construido un vínculo sobrepuesto de *B* a *D*, en vez de hacerlo de *A* a *C*. Tal configuración hubiese evitado el atravesar dos veces por los vínculos <*Ra*, *Rb*> y <*Rc*, *Rd*>.

La calidad de un árbol de multitransmisión al nivel de aplicación se mide, por lo general, mediante tres métricas diferentes: tensión del vínculo, estiramiento, y costo del árbol. La **tensión del vínculo** se define por vínculo y mide la frecuencia con que un paquete cruza el mismo vínculo (Chu y cols., 2002). Una tensión de vínculo mayor a 1 proviene de que aunque a un nivel lógico un paquete puede reenviarse a lo largo de dos conexiones diferentes, parte de esas conexiones pueden en realidad corresponder al mismo vínculo físico, como ilustra la figura 4-31.

El **estiramiento**, o **Castigo Relativo por Retraso** (**RDP**, por sus siglas en inglés), mide la relación que hay en el retraso entre dos nodos de la sobrepuesta, y el retraso que esos dos nodos experimentarían en la red subyacente. Por ejemplo, en la red sobrepuesta, de *B* a *C* los mensajes siguen la ruta *B* → *Rb* → *Ra* → *Rc* → *C*, y tienen un costo total de 59 unidades. Sin embargo, los mensajes hubiesen podido enrutarse en la red subyacente a lo largo de la ruta *B* → *Rb* → *Rd* → *Rc* → *C*, con un costo total de 47 unidades, lo cual arroja un estiramiento de 1.255. Desde luego, cuando se construye una red sobrepuesta, el objetivo es minimizar el estiramiento agregado o, de modo similar, el RDP promedio medido sobre todos los pares de nodos.

Por último, el **costo del árbol** es una métrica global relacionada, generalmente, con la minimización de costos agregados a los vínculos. Por ejemplo, si el costo de un vínculo se toma como el

retraso entre sus dos nodos finales, entonces optimizar el costo del árbol se reduce para encontrar un árbol de longitud mínima en la que el tiempo total para diseminar la información a todos los nodos es mínima.

Para simplificar las cosas de alguna manera, supongamos que un grupo de multitransmisión tiene un nodo asociado muy conocido, el cual rastrea a los nodos que se han unido al árbol. Cuando un nuevo nodo emite una petición de unión, contacta a su **nodo rendezvous** (nodo central) para obtener una lista (potencialmente parcial) de miembros. El objetivo es seleccionar al mejor miembro para operar como el padre del nuevo nodo del árbol. ¿Quién debe seleccionarlo? Existen muchas alternativas y propuestas distintas que con frecuencia arrojan diferentes soluciones.

Por ejemplo, considere un grupo de multitransmisión con una sola fuente. En este caso, la selección del mejor nodo es evidente: debe ser la fuente (ya que en ese caso podemos estar seguros de que el estiramiento será igual a 1). Sin embargo, al hacerlo, introduciríamos una topología estrella con la fuente en el centro. Aunque simple, no es difícil imaginar que la fuente puede fácilmente resultar sobrecargada. En otras palabras, la selección de un nodo generalmente estará restringida de tal manera que sólo aquellos nodos con k o menos vecinos pueden ser elegidos, donde k es un parámetro de diseño. Esta restricción complica severamente el algoritmo para establecer el árbol, cuando una buena solución puede requerir que parte del árbol existente se reconfigure.

Tan y colaboradores (2003) proporcionan un amplio panorama y una evaluación a varias soluciones para este problema. Como ilustración, consideremos una familia específica, conocida como **árboles de intercambio** (Helder y Jamin, 2002). La idea básica es sencilla. Suponga que ya tenemos un árbol de multitransmisión con una sola fuente como raíz. En este árbol, un nodo P puede intercambiar padres, arrastrando el vínculo hacia su padre actual, a favor de un vínculo con otro nodo. Las únicas restricciones impuestas para el intercambio de vínculos son que el nuevo padre nunca pueda ser un miembro del subárbol con raíz en P (ya que partiría el árbol y crearía un ciclo), y que el nuevo padre no tendrá demasiados hijos directos. Esto último es necesario para limitar la carga de reenvío de mensajes por un solo nodo.

Existen diferentes criterios para decidir el intercambio de padres. Un enfoque sencillo es optimizar la ruta hacia la fuente, lo cual minimiza de manera efectiva el retraso cuando un mensaje va a multitransmitirse. Con este fin, cada nodo recibe regularmente información de otros nodos (más adelante explicaremos una forma específica para hacerlo). En ese punto, el nodo puede evaluar si otro nodo sería un mejor padre en términos del retraso a través de la ruta hacia la fuente y, si lo es, iniciar el intercambio.

Otro criterio podría ser si el retraso del otro padre en potencia es menor que el del padre actual. Si cada nodo toma este criterio, entonces los retrasos agregados del árbol resultante deben ser idealmente mínimos. En otras palabras, éste es un ejemplo de cómo optimizar el costo del árbol tal como explicamos antes. Sin embargo, se necesitaría más información para construir un árbol similar, pero como resulta evidente, este sencillo esquema representa una heurística razonable que nos conduce a una buena aproximación de un árbol de longitud mínima.

A manera de ejemplo, consideremos el caso en que un nodo P recibe información sobre los vecinos de su padre. Observe que los vecinos consisten en los abuelos de P, junto con los demás hermanos de P. El nodo P puede evaluar entonces los retrasos de cada uno de estos nodos, y posteriormente elegir al que tenga el retraso más bajo, digamos Q, como su nuevo padre. Con este fin, envía a Q una petición de intercambio. Para evitar que se formen ciclos debido a peticiones

concurrentes de intercambio, un nodo que tiene una petición de intercambio pendiente simplemente se negará a procesar cualquier petición de entrada. En efecto, esto conduce a una situación en la que únicamente los intercambios completamente independientes pueden llevarse a cabo de manera simultánea. Más aún, *P* proporcionará a *Q* información suficiente para permitir que este último concluya que ambos nodos tienen el mismo padre, o que *Q* es el abuelo.

Un problema importante que aún no hemos abordado es la falla de un nodo. En el caso de árboles de intercambio, se propone una solución sencilla: siempre que un nodo descubra que su padre ha fallado, simplemente se adjunta a la raíz. En ese punto, el protocolo de optimización procede en la forma usual, y en algún momento coloca al nodo en un buen punto del árbol de multitransmisión. Los experimentos descritos en Helder y Jamin (2002) muestran que el árbol resultante es, de hecho, parecido a uno de longitud mínima.

4.5.2 Diseminación de datos basada en el gossip

Una técnica que cobra cada vez más importancia para implementar la diseminación de información se basa en el *comportamiento epidémico*. Al observar cómo se propagan las enfermedades entre la gente, los investigadores han estudiado desde hace mucho si técnicas tan simples pudieran desarrollarse para propagar información en sistemas distribuidos de gran escala. El objetivo principal de estos **protocolos epidémicos** es propagar rápidamente información entre una gran colección de nodos utilizando sólo información local. En otras palabras, no hay un componente central mediante el cual se coordine la diseminación de información.

Para explicar los principios generales de estos algoritmos, suponemos que para un elemento específico de datos, todas las actualizaciones se inician en un solo nodo. De esta manera, simplemente evitamos conflictos escritura-escritura. La siguiente presentación se basa en el clásico artículo de Demers y colaboradores, (1987), sobre algoritmos epidémicos. Una visión general reciente sobre diseminación epidémica de información puede encontrarse en Eugster y colaboradores, (2004).

Modelos de diseminación de información

Tal como sugiere su nombre, los algoritmos epidémicos se basan en la teoría de las epidemias, la cual estudia la propagación de enfermedades infecciosas. En el caso de sistemas distribuidos de gran escala, en lugar de enfermedades infecciosas propaga información. El estudio de epidemias para sistemas distribuidos también apunta hacia un objetivo completamente diferente: mientras que las organizaciones de salud harán su mayor esfuerzo para prevenir que las enfermedades infecciosas se propaguen entre grandes grupos de personas, los diseñadores de algoritmos epidémicos para sistemas distribuidos intentarán "infectar" todos los nodos con la nueva información tan rápidamente como sea posible.

Si utilizamos la terminología de epidemias, decimos que un nodo integrante de un sistema distribuido está **infectado** cuando mantiene información que vale la pena propagar hacia otros nodos. Un nodo que aún no ha visto esta información se conoce como **susceptible**. Por último, se dice que

un nodo actualizado que no esté dispuesto o no sea capaz de propagar su información será **eliminado**. Observe que suponemos que podemos distinguir la información vieja de la nueva, por ejemplo, porque se le ha asignado un registro de tiempo o un número de versión. En este sentido, se afirma que los nodos también propagan actualizaciones.

Un popular modelo de propagación es el de **antientropía**. En este modelo, un nodo *P* elige al azar a otro nodo *Q*, y posteriormente intercambia actualizaciones con *Q*. Existen tres métodos para intercambiar actualizaciones:

1. *P* sólo empuja sus propias actualizaciones hacia *Q*
2. *P* sólo jala nuevas actualizaciones desde *Q*
3. *P* y *Q* se envían actualizaciones entre sí (es decir, un método del tipo jala-empuja)

Cuando se trata de propagar actualizaciones rápidamente, el sólo empujarlas parece una mala alternativa. Por intuición, esto puede interpretarse de la siguiente forma. Primero, observe que en un método basado puramente en empujar, las actualizaciones pueden propagarse utilizando únicamente nodos infectados. Sin embargo, si hay muchos nodos infectados, la probabilidad de que cada uno seleccione a un nodo susceptible es relativamente pequeña. En consecuencia, hay posibilidades de que un nodo en particular permanezca susceptible durante un periodo largo simplemente porque no es seleccionado por un nodo infectado.

Por contraste, el método basado en jalar funciona mucho mejor cuando muchos nodos están infectados. En ese caso, la propagación de actualizaciones es disparada básicamente por nodos susceptibles. Hay buenas posibilidades de que un nodo contacte a uno infectado para que posteriormente jale las actualizaciones y se vuelva también uno infectado.

Es posible demostrar que si únicamente está infectado un nodo, las actualizaciones se propagarán rápidamente a todos los nodos utilizando cualquier forma de antientropía, aunque el jalar-empujar permanezca como la mejor estrategia (Jelasity y cols., 2005a). Una **ronda** se define como un periodo en el que todos los nodos han tomado, al menos una vez, la iniciativa de intercambiar actualizaciones con algún otro nodo elegido al azar. Es posible demostrar entonces que el número de rondas necesarias para propagar una sola actualización hacia todos los nodos es $O(log(N))$, donde N es la cantidad de nodos presentes en el sistema. De hecho, esto indica que propagar actualizaciones es rápido, pero sobre todo escalable.

Una variante específica de este método se conoce como **propagación de rumores**, o simplemente **gossiping**. El gossiping funciona de la siguiente manera. Si el nodo *P* se acaba de actualizar con el elemento de datos *x*, contacta a cualquier otro nodo *Q* e intenta empujar la actualización en *Q*. Sin embargo, es posible que *Q* ya haya sido actualizado por otro nodo. En ese caso, *P* podría perder el interés en propagar la actualización nuevamente, digamos, en una probabilidad de $1/k$. En otras palabras, la actualización se elimina entonces.

El gossiping es completamente análogo a la vida real. Cuando Bob tiene chismes que contar, tal vez llame por teléfono a su amiga Alice para contarle qué sucede. Alice, al igual que Bob, se sentirá ansiosa por propagar el chisme hacia sus amigos. Sin embargo, se sentirá desilusionada si llama

a uno de ellos, digamos a Chuck, y sólo escucha que él ya lo sabía. Las posibilidades son que dejará de llamar a otros amigos, pues ¿para qué les llama si ya saben el chisme?

El gossiping resulta ser una excelente forma de propagar noticias. Sin embargo, no hay garantía de que todos los nodos serán actualizados (Demers y cols., 1987). Es posible demostrar que cuando hay una gran cantidad de nodos participantes de la epidemia, la fracción s de nodos que permanecerá ignorante de una actualización, es decir, se conservará susceptible, satisface la ecuación:

$$s = e^{-(k+1)(1-s)}$$

La figura 4-32 muestra a $ln(s)$ como una función de k. Por ejemplo, si $k = 4$, $ln(s) = -4.97$, por lo que s es menor que 0.007, ello significa que menos del 0.7% de los nodos permanece susceptible. No obstante, se necesitan medidas especiales para garantizar que dichos nodos también se actualizarán. Al combinar la antientropía con el gossiping se logra el truco.

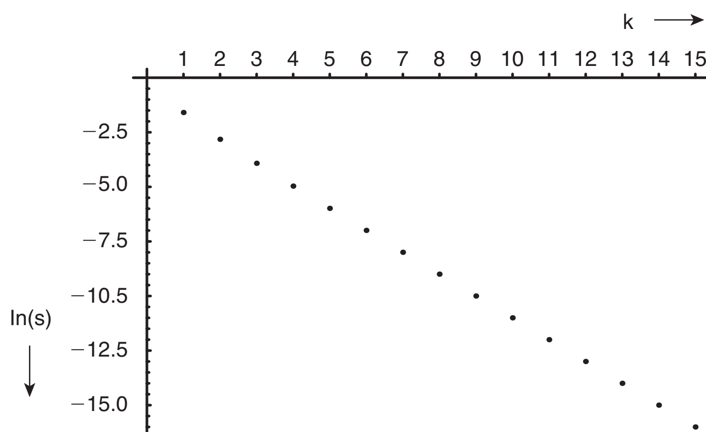


Figura 4-32. Relación entre la fracción s de los nodos que ignoran la actualización y el parámetro *k* del gossiping. La gráfica despliega el *ln*(*s*) como una función de *k*.

Una de las principales ventajas de los algoritmos epidémicos es su escalabilidad, debido a que el número de sincronizaciones entre procesos es relativamente pequeño comparado con otros métodos de propagación. Para sistemas de área amplia, Lin y Marzullo (1999) muestran que tiene sentido considerar la topología real de la red para lograr mejores resultados. En su método, los nodos conectados sólo a algunos cuantos nodos son contactados con una probabilidad relativamente alta. La suposición subyacente es que tales nodos forman un puente hacia otras partes remotas de la red; por tanto, deben ser contactados tan pronto como sea posible. Este método es conocido como **gossiping direccional** y se presenta en diferentes variantes.

Este problema plantea una suposición importante que hacen la mayoría de las soluciones, a saber: un nodo puede seleccionar al azar a cualquier otro nodo con el cual "chismosear". Esto implica

que, en principio, todo el grupo de nodos debe conocer a cada miembro. En un sistema grande, esta suposición no tiene bases.

Por fortuna, no hay necesidad de tener una lista. Como explicamos en el capítulo 2, mantener una visión parcial actualizada más o menos continuamente organiza al grupo de nodos en una gráfica aleatoria. Si regularmente se actualiza la vista parcial de cada nodo, la selección aleatoria ya no resulta ser un problema.

Eliminación de datos

Los algoritmos epidémicos son extremadamente buenos para propagar actualizaciones. Sin embargo, tienen un extraño efecto colateral: propagar la *eliminación* de un elemento de datos es difícil. La esencia del problema radica en que la eliminación de un elemento de datos destruye toda la información de ese elemento. En consecuencia, cuando un elemento de datos es eliminado de un nodo, dicho nodo recibirá en algún momento viejas copias del elemento, y las interpretará como actualizaciones de algo que no tenía antes.

El truco es registrar la eliminación de un elemento de datos como otra actualización cualquiera, y mantener un registro de dicha eliminación. De esta manera, las viejas copias no se interpretarán como algo nuevo, sino tan sólo como versiones actualizadas mediante una operación de eliminación. El registro de una eliminación se lleva a cabo gracias a la propagación de **certificados de defunción**.

Por supuesto, el problema con los certificados de defunción es que deben limpiarse en algún momento, ya que de otra forma cada nodo construiría gradualmente una gran base de datos local con información histórica de elementos eliminados. Demers y colaboradores (1987) proponen utilizar lo que llaman certificados de defunción latentes. A cada certificado de defunción se le asigna un registro de tiempo cuando se crea. Si es posible suponer que las actualizaciones se propagarán a todos los nodos dentro de un tiempo finito conocido, entonces los certificados de defunción pueden eliminarse después que este tiempo de propagación máxima haya pasado.

Sin embargo, para proporcionar garantías sólidas de que las eliminaciones se propagarán hacia todos los nodos, sólo unos cuantos nodos mantienen certificados de defunción latentes que nunca se desechan. Supongamos que el nodo P tiene uno de estos certificados para el elemento x. Si por algún motivo una actualización obsoleta para x llega a P, éste reaccionará con la simple propagación del certificado de defunción para x.

Aplicaciones

Para finalizar esta presentación, veamos algunas aplicaciones interesantes de los protocolos epidémicos. Ya mencionamos la propagación de actualizaciones, que tal vez sea la aplicación más ampliamente utilizada. Además, en el capítulo 2 explicamos cómo es que proporcionar información sobre la posición de los nodos puede ayudar a construir topologías específicas. De la misma forma, el gossiping puede utilizarse para descubrir nodos que tienen algunos vínculos de área amplia de salida, para posteriormente aplicar el gossiping direccional como ya mencionamos.

Otra área de aplicación interesante es la de simplemente recopilar o, de hecho, agregar información (Jelasity y cols., 2005b). Consideremos el siguiente intercambio de información. Cada

nodo i elige inicialmente un número cualquiera, digamos x_i. Cuando el nodo i contacta al nodo j, cada uno actualiza su valor como:

$$x_i, x_j \leftarrow (x_i + x_j) / 2$$

Desde luego, después de este intercambio, tanto i como j tendrán el mismo valor. De hecho, no es difícil advertir que en algún momento todos los nodos tendrán el mismo valor, a saber, el promedio de todos los valores iniciales. La velocidad de propagación de nuevo resulta ser exponencial.

¿Qué uso tiene el promedio calculado? Consideremos la situación en que todos los nodos i han inicializado en cero a x_i, excepto x_1 que lo ha inicializado en 1:

$$x_i \quad \begin{cases} 1 \text{ si } i = 1 \\ 0 \text{ si } i > 1 \end{cases}$$

Si hay N nodos, en algún momento cada nodo calculará entonces el promedio, que es $1/N$. En consecuencia, cada nodo i puede estimar el tamaño del sistema como $1/x_i$. Esta información puede utilizarse para ajustar dinámicamente varios parámetros del sistema. Por ejemplo, el tamaño de la vista parcial (es decir, el número de vecinos que cada nodo rastrea) debe depender del número total de nodos participantes. Conocer este número permitirá que un nodo ajuste dinámicamente el tamaño de su vista parcial. De hecho, esto puede considerarse como una propiedad de autoadministración.

Calcular el promedio puede resultar difícil cuando los nodos entran y salen del sistema con regularidad. Una solución práctica para este problema es introducir épocas. Si suponemos que el nodo 1 es estable, éste simplemente inicia una nueva época ahora y después. Cuando el nodo i ve una nueva época por primera vez, reinicia su propia variable x_i en cero, e inicia nuevamente el cálculo del promedio.

Desde luego, también es posible obtener otros resultados. Por ejemplo, en lugar de hacer que un nodo fijo (x_1) inicie el cálculo del promedio, podemos elegir fácilmente un nodo al azar en la siguiente forma. Cada nodo i establece inicialmente a x_i en un número aleatorio del mismo intervalo, digamos [0, 1], y también lo almacena permanentemente como m_j. Además de un intercambio entre los nodos i y j, cada uno cambia su valor a:

$$x_i, x_j \leftarrow máx(x_i = x_j)$$

Cada nodo i para el que $m_i < x_i$ perderá la competencia por ser el iniciador del cálculo del promedio. Al final, habrá un solo ganador. Por supuesto, aunque resulta muy sencillo concluir que un nodo ha perdido, es mucho más difícil decidir que ha ganado porque es incierto si los resultados han llegado. La solución a este problema es ser optimista: un nodo siempre asume que es el ganador hasta que se demuestre lo contrario. En ese punto, la variable utilizada para calcular el promedio simplemente reinicia en cero. Observe que para entonces muchos otros cálculos (en nuestro ejemplo, el cálculo de un máximo y el de un promedio) pueden estar en ejecución simultáneamente.

4.6 RESUMEN

Contar con herramientas poderosas y flexibles para lograr la comunicación entre procesos es esencial para cualquier sistema distribuido. En aplicaciones de red tradicionales, la comunicación con frecuencia se basa en las primitivas de bajo nivel para el paso de mensajes que ofrece la capa de transporte. Un tema importante en los sistemas middleware es ofrecer un nivel más alto de abstracción que haga más sencilla la comunicación entre procesos que el soporte ofrecido por la interfaz para la capa de transporte.

Una de las abstracciones más ampliamente utilizadas es la Llamada a Procedimientos Remotos (RPC). La esencia de una RPC es que un servicio se implementa por medio de un procedimiento, cuyo cuerpo es ejecutado en un servidor. Al cliente sólo se le ofrece la firma del procedimiento, es decir, el nombre del procedimiento junto con sus parámetros. Cuando el cliente llama al procedimiento, la implementación del lado del cliente, llamado resguardo, se encarga de guardar los valores de los parámetros en un mensaje y de enviar éste al servidor. El servidor llama al procedimiento real y devuelve los resultados, de nuevo en un mensaje. El resguardo del cliente extrae los valores resultantes del mensaje devuelto, y nuevamente lo pasa a la aplicación cliente que realizó la llamada.

Las RPC ofrecen herramientas para implementar la comunicación síncrona, mediante la cual un cliente es bloqueado hasta que el servidor envía una respuesta. Aunque existen variantes del mecanismo mediante el cual este estricto modelo síncrono se relaja, se vuelve evidente que los modelos de alto nivel orientados a mensajes y de propósito general con frecuencia resultan más convenientes.

En modelos orientados a mensajes, la cuestión es si la comunicación resulta o no persistente, y si es o no síncrona. La esencia de la comunicación persistente es que un mensaje presentado para transmisión sea almacenado por el sistema de comunicación hasta que logre entregarlo. En otras palabras, ni el remitente ni el destinatario necesitan estar en ejecución para que ocurra la transmisión de un mensaje. En la comunicación transitoria no se ofrecen herramientas de almacenamiento, por lo que el destinatario debe estar preparado para aceptar el mensaje cuando le sea enviado.

En comunicación asíncrona, al remitente se le permite continuar de inmediato después que el mensaje se presentó para su transmisión, probablemente antes de enviarlo. En comunicación síncrona, el remitente es bloqueado al menos hasta que el mensaje se recibe. De modo alterno, el remitente puede ser bloqueado hasta que ocurra la entrega del mensaje, o hasta que el destinatario responda con una RPC.

Los modelos middleware orientados a mensajes, por lo general, ofrecen comunicación asíncrona persistente y se utilizan cuando las RPC no son adecuadas. Con frecuencia, se utilizan para ayudar en la integración de colecciones de bases de datos (ampliamente dispersas), en sistemas de información de gran escala. Otras aplicaciones incluyen correo electrónico y volúmenes de trabajo.

Una manera muy diferente de comunicación es la del flujo continuo, en la que el asunto es si dos mensajes sucesivos tienen o no una relación temporal. En flujos continuos de datos, se especifica un retraso máximo fin a fin para cada mensaje. Además, también es necesario que los mensajes enviados estén sujetos a un retraso mínimo fin a fin. Ejemplos típicos de tales flujos continuos

de datos son los de video y audio. Lo que son exactamente de relación temporal, o lo que se espera del subsistema subyacente de comunicación en términos de la calidad del servicio, con frecuencia es difícil de especificar e implementar. Un factor que complica las cosas es el papel de la inestabilidad. Incluso si el rendimiento promedio es aceptable, variaciones importantes en el tiempo de entrega pueden derivar en un rendimiento inaceptable.

Por último, en sistemas distribuidos, una clase importante de protocolos de comunicación es la multitransmisión. La idea básica es diseminar información de un remitente hacia diversos destinatarios. Hemos explicado dos métodos diferentes. Primero, la multitransmisión puede lograrse al configurar un árbol del remitente hacia los destinatarios. Si consideramos que ahora comprendemos bien cómo es que los nodos pueden autoorganizarse en sistemas de punto a punto, también han aparecido soluciones para configurar dinámicamente árboles de manera descentralizada.

Otra clase importante de soluciones de diseminación utiliza protocolos epidémicos. Estos protocolos han demostrado ser muy sencillos, aunque demasiado grandes. Además de sólo diseminar mensajes, los protocolos epidémicos también pueden utilizarse eficientemente para agregar información a través de un gran sistema distribuido.

PROBLEMAS

1. En muchos protocolos de capas, cada capa tiene su propio encabezado. De seguro resultaría más eficiente tener un solo protocolo al frente de cada mensaje con todo el control centrado en él que todos esos encabezados por separado. ¿Por qué no se implementa esto?

2. ¿Por qué los servicios de comunicación al nivel de transporte con frecuencia resultan inadecuados para construir aplicaciones distribuidas?

3. Un servicio confiable de multitransmisión permite al remitente pasar mensajes de manera confiable a una serie de destinatarios. ¿Tal servicio pertenece a una capa middleware o debe ser parte de una capa de más bajo nivel?

4. Considere un procedimiento *incr* con dos parámetros enteros. El procedimiento agrega uno a cada parámetro. Ahora suponga que éste es llamado con la misma variable dos veces, por ejemplo, como *incr*(*i, i*). Si se inicializa a *i* con 0, ¿qué valor tendrá después *i* cuando se utiliza una llamada por referencia? ¿Qué sucede cuando se utiliza una llamada por copia-restauración?

5. C tiene una construcción llamada unión, en la que el campo de un registro (llamado *struct* en C) puede mantener cualquiera de varias alternativas. En tiempo de ejecución, no hay una manera segura de saber cuál alternativa se encuentra ahí. ¿Esta característica de C tiene alguna implicación para las llamadas a procedimientos remotos? Explique su respuesta.

6. Una manera de manejar la conversión de parámetros en sistemas RPC es hacer que cada máquina envíe parámetros en su representación y que otra haga la traducción, si es necesario. El sistema original podría señalarse mediante un código insertado en el primer byte. Sin embargo, debido a que localizar el primer byte de la primera palabra es precisamente el problema, ¿puede funcionar esto?

7. Suponga que un cliente llama a una RPC asíncrona para un servidor, y posteriormente espera hasta que el servidor devuelve un resultado utilizando otra RPC asíncrona. ¿Es este método lo mismo que dejar que el cliente ejecute una RPC normal? ¿Qué sucede si reemplazamos las RPC asíncronas por RPC asíncronas?

8. En lugar de permitir que un servidor se registre a sí mismo con un demonio como en DCE, también podríamos elegir asignarle siempre el mismo punto final. El punto final puede entonces utilizarse en referencias a objetos en el espacio de dirección del servidor. ¿Cuál es la principal desventaja de este esquema?

9. ¿Resultaría útil diferenciar RPC dinámicas y estáticas?

10. Describa cómo sucede la comunicación orientada a no conexión entre un cliente y un servidor cuando se utilizan sockets.

11. Explique la diferencia entre las primitivas MPI_bsend y MPI_isend en MPI.

12. Suponga que sólo pueden utilizarse primitivas de comunicación asíncrona transitoria, incluyendo una sola primitiva de recepción asíncrona. ¿Cómo implementaría usted primitivas para comunicación *síncrona* transitoria?

13. Suponga que sólo puede utilizar primitivas de comunicación síncrona transitoria. ¿Cómo implementaría primitivas para comunicación *asíncrona* transitoria?

14. ¿Tiene sentido implementar la comunicación asíncrona persistente utilizando RPC?

15. En el texto establecimos que para iniciar automáticamente un proceso para encontrar mensajes en una cola de entrada se utiliza, con frecuencia, un demonio que da seguimiento a la cola de entrada. Proponga una implementación alterna que no utilice un demonio.

16. Las tablas de enrutamiento en IBM WebSphere, y en muchos otros sistemas de colas de mensajes, se configuran manualmente. Describa una forma simple para hacer esto de modo automático.

17. Con comunicación persistente, por lo general un destinatario tiene su propio bufer local donde los mensajes pueden almacenarse cuando no está en ejecución. Para crear un bufer como éste deberemos especificar su tamaño. Proponga un argumento sobre por qué es preferible hacer esto, así como uno en contra de especificar el tamaño.

18. Explique por qué la comunicación síncrona transitoria tiene problemas inherentes de escalabilidad, y cómo podrían resolverse tales problemas.

19. Proporcione un ejemplo en el que la multitransmisión también sea útil para tratar con flujos discretos de datos.

20. Suponga que en una red de sensores, las temperaturas medidas no son registradas en el tiempo por el sensor, sino que se envían inmediatamente al operador. ¿Esto sería suficiente para garantizar sólo un retraso máximo fin a fin?

21. ¿Cómo podría garantizar usted un retraso máximo fin a fin cuando una serie de computadoras está organizada en anillo (lógico o físico)?

22. ¿Cómo podría garantizar usted un retraso mínimo fin a fin cuando una serie de computadoras está organizada en anillo (lógico o físico)?

23. A pesar de que la multitransmisión es técnicamente factible, hay muy poco soporte para utilizarla en internet. La respuesta a este problema es observar los modelos de negocios ya aterrizados: nadie sabe realmente cómo hacer dinero a partir de la multitransmisión. ¿Qué esquema puede proponer usted?

24. En general, los árboles de multitransmisión al nivel de aplicaciones se optimizan con respecto a la elasticidad, la cual se mide en términos de retrasos. Proporcione un ejemplo en el que esta métrica podría derivar en árboles muy malos.

25. Cuando se buscan archivos en un sistema no estructurado de punto a punto, podría resultar útil restringir la búsqueda a los nodos que tienen archivos parecidos a los suyos. Explique cómo el gossiping puede ayudarle a encontrar tales nodos.

5

NOMBRES

Los nombres juegan un papel muy importante en todos los sistemas de cómputo. Se utilizan para compartir recursos, para identificar entidades de manera única, para hacer referencia a ubicaciones, y más. Un asunto importante con respecto a los nombres es que un nombre puede resolverse para la entidad a la cual hace referencia. De esta manera, la resolución de nombres permite a un proceso acceder a la entidad nombrada. Para resolver los nombres, es necesario implementar un sistema de nombres. En sistemas distribuidos y sistemas no distribuidos, la diferencia entre nombres radica en la forma en que se implementan los sistemas.

En un sistema distribuido, con frecuencia la implementación de un sistema de nombres se distribuye por sí sola a través de múltiples máquinas. La manera en que se lleva a cabo esta distribución juega un papel importante en lo que se refiere a la eficiencia y la escalabilidad del sistema de nombres. En este capítulo, nos concentramos en tres diferentes e importantes formas de utilizar los nombres en los sistemas distribuidos.

Primero, después de explicar algunos temas generales relacionados con los nombres, daremos un vistazo a la organización e implementación de nombres amigables con el usuario. Ejemplos típicos de dichos nombres incluyen aquellos implementados para los sistemas de archivos y para la World Wide Web. La construcción de sistemas de nombres escalables a nivel mundial es una preocupación primordial para este tipo de nombres.

Segundo, los nombres se utilizan para ubicar a las entidades en una vía independiente de su ubicación actual. Como podemos darnos cuenta, los sistemas de nombres para nombres amigables con el usuario no están configurados de manera particular para soportar este tipo de entidades de seguimiento. La mayoría de los nombres ni siquiera dan clave alguna con respecto a la ubicación de la

entidad. Se requieren organizaciones alternativas tales como las utilizadas para la telefonía móvil, en donde los nombres son identificadores independientes de la ubicación, y aquellas empleadas para las tablas hash distribuidas.

Por último, con frecuencia las personas prefieren describir entidades por medio de distintas características, ello provoca una situación en la cual, necesitamos resolver una descripción mediante los atributos apropiados para una entidad que se adhiera a dicha descripción. Este tipo de resolución de nombres es muy difícil y le pondremos atención especial.

5.1 NOMBRES, IDENTIFICADORES Y DIRECCIONES

Comencemos por dar un vistazo a lo que actualmente es un nombre. Un nombre —dentro de un sistema distribuido— es una cadena de bits o caracteres utilizados para hacer referencia a una entidad. En un sistema distribuido, una entidad puede ser prácticamente cualquier cosa. Ejemplos clásicos incluyen recursos tales como servidores, impresoras, discos y archivos. Otros ejemplos muy conocidos de entidades que a menudo se nombran de manera explícita son los procesos, usuarios, buzones de correo, grupos de noticias, páginas web, ventanas gráficas, mensajes, conexiones de red y muchas cosas más.

Las entidades se pueden operar. Por ejemplo, un recurso tal como una impresora ofrece una interfaz que contiene operaciones para imprimir un documento, solicitar el estado de una tarea de impresión y cosas similares. Además, una entidad tal como una conexión de red puede proporcionar operaciones para enviar y recibir datos, establecer parámetros de calidad de servicio, solicitar el estado y así sucesivamente.

Para operar una entidad, es necesario acceder a ella, de modo que se requiere tener un **punto de acceso**. Un punto de acceso es otra clase, pero especial, de entidad en un sistema distribuido. Al nombre de un punto de acceso se le llama **dirección**. A la dirección del punto de acceso de una entidad se le denomina dirección de dicha entidad.

Una entidad puede ofrecer más de un punto de acceso. Como comparación, un teléfono puede verse como un punto de acceso a una persona, en donde el número telefónico corresponde a una dirección. De hecho, en la actualidad mucha gente tiene distintos números de teléfono, y cada número corresponde a un punto en donde se le puede localizar. En un sistema distribuido, un ejemplo típico de un punto de acceso es un servidor que ejecuta un servidor específico, en el cual la dirección está formada por, digamos, una dirección IP y un número de puerto (es decir, la dirección del servidor a nivel de transporte).

Una entidad puede modificar sus puntos de acceso en el curso del tiempo. Por ejemplo, cuando una computadora móvil se mueve a una ubicación diferente, con frecuencia se le asigna una IP distinta a la que tenía antes. De manera similar, cuando una persona se traslada a otra ciudad u otro país, a menudo es necesario modificar también sus números telefónicos. De igual modo, cambiar de empleo o de proveedor de servicios de internet significa modificar nuestra dirección de correo electrónico.

Entonces, una dirección es sólo una clase especial de nombre: hace referencia a un punto de acceso de una entidad. Debido a que un punto de acceso está íntimamente asociado con una entidad, parecería conveniente utilizar la dirección de un punto de acceso como nombre regular de la entidad asociada. Sin embargo, esto difícilmente se lleva a cabo dado que la nomenclatura es, por lo general, muy inflexible y a menudo nada amigable con el usuario.

Por ejemplo, no es poco común reorganizar un sistema distribuido con regularidad, de manera que un servidor específico pueda ejecutarse entonces en un servidor diferente al previo. La máquina anterior sobre la cual solía ejecutarse se puede reasignar a un servidor completamente diferente. En otras palabras, una entidad puede modificar fácilmente un punto de acceso, o un punto de acceso puede reasignarse a una entidad diferente. Si una dirección se utiliza para hacer referencia a una entidad, tendremos una referencia inválida en el instante en que el punto de acceso cambie o sea reasignado a otra entidad. Por tanto, es mucho mejor dejar que un servicio sea conocido por un nombre separado independientemente de la dirección del servidor asociado.

De modo similar, si una entidad ofrece más de un punto de acceso, no queda claro cuál dirección utilizar como referencia. Por ejemplo, muchas organizaciones distribuyen su servicio web a través de muchos servidores. Si utilizamos las direcciones asignadas a dichos servidores como referencia para un servicio web, no resulta evidente qué dirección se debe elegir como la mejor. De nuevo, una mucho mejor solución es la de tener un solo nombre para el servicio web independientemente de las direcciones de los diferentes servidores web.

Estos ejemplos ilustran que un nombre para una entidad que es independiente de sus direcciones con frecuencia es más fácil y más flexible de usar. A dicho nombre se le conoce como **independiente de su ubicación**.

Además de la dirección, existen otros tipos de nombres que merecen un trato especial, tales como los nombres empleados para identificar de manera única a una entidad. Un **identificador** verdadero es un nombre que tiene las propiedades siguientes (Wieringa y de Jonge, 1995):

1. Un identificador hace referencia a una entidad como máximo.
2. Cada entidad es referida por al menos un identificador.
3. Un identificador siempre hace referencia a la misma entidad (es decir, nunca se reutiliza).

Mediante el uso de identificadores se vuelve más fácil hacer referencia clara a una entidad. Por ejemplo, asumamos que dos procesos hacen referencia a una entidad por medio de un identificador. Para verificar si los procesos hacen referencia a la misma entidad, es suficiente con verificar que los dos identificadores sean iguales. Dicha prueba no es suficiente si los dos procesos utilizan nombres regulares, no únicos, y no identificables. Por ejemplo, el nombre "Juan Pérez" no se puede tomar como una referencia única para una sola persona.

De manera similar, si se puede reasignar una dirección a una entidad diferente, no podemos utilizar una dirección como un identificador. Consideremos el uso de números telefónicos, los cuales son razonablemente estables en el sentido de que un número telefónico hace referencia durante cierto tiempo a la misma persona o empresa. Sin embargo, usar un número telefónico como identificador no funcionará, ya que puede ser reasignado en el curso del tiempo. En consecuencia, la pastelería de Bob pudiera recibir llamadas telefónicas para la tienda de antigüedades de Alicia durante un largo periodo. En este caso, hubiera sido mejor utilizar un identificador verdadero para Alicia en lugar de su número telefónico.

Las direcciones y los identificadores son dos importantes tipos de nombres empleados cada uno con propósitos muy diferentes. En muchos sistemas de cómputo, las direcciones y los identificadores se representan usando solamente formas legibles para la máquina, esto es, en forma de cadenas

de bits. Por ejemplo, una dirección ethernet es, en esencia, una cadena aleatoria de 48 bits. De manera similar, las direcciones de memoria se representan por lo general como cadenas de 32 o 64 bits.

Otro tipo importante de nombre es el destinado a utilizarse por las personas, también llamado **nombre amigable para el usuario**. Al contrario de las direcciones y los identificadores, un nombre amigable para el usuario por lo general, se representa como una cadena de caracteres. Estos nombres aparecen de muchas formas diferentes. Por ejemplo, en los sistemas UNIX los archivos tienen cadenas de caracteres como nombres que pueden tener una longitud de hasta 255 caracteres, y los cuales son definidos completamente por el usuario. De manera similar, los nombres DNS están representados como sencillas cadenas de caracteres sensibles a mayúsculas y minúsculas.

Tener nombres, identificadores, y direcciones nos da el tema central del presente capítulo, ¿cómo resolver nombres e identificadores de direcciones? Antes de abordar distintas soluciones, es importante darnos cuenta de que existe a menudo una relación cercana entre la resolución de nombres en los sistemas distribuidos y el ruteo de mensajes. En principio, un sistema de nombres mantiene un **vínculo nombre a dirección** el cual es, en su forma más simple, solamente una tabla (*nombre, dirección*) de pares. Sin embargo, en sistemas distribuidos que se expanden en grandes redes y para las cuales se requieren muchos recursos, una tabla centralizada no funcionará.

Al contrario, sucede con frecuencia que el nombre se descompone en muchas partes tales como *ftp.cs.vu.nl*, y dicha resolución de nombres tiene lugar a través de una búsqueda recursiva de dichas partes. Por ejemplo, un cliente que requiere saber la dirección del servidor FTP relacionado con *ftp.cs.vu.nl* primero resolverá *nl* para encontrar al servidor *NS(nl*) responsable de los nombres que terminan con *nl*, después de lo cual se pasa el resto del nombre al servidor *NS(nl)*. Entonces, este servidor puede resolver el nombre *vu* en el servidor *NS*(*vu.nl*) responsable de los nombres que terminan con *vu.nl* los cuales pueden manipular el resto del nombre *ftp.cs*. En algún punto, esto provoca el ruteo de la resolución de nombres como:

Ns(.) → *NS(nl)* → *NS(vu.nl)* → dirección de *ftp.cs.vu.nl*

donde *NS(.)* representa al servidor que puede devolver la dirección de *NS*(*nl*), también conocido como **servidor raíz**. *NS*(*vu.nl*) devolverá la dirección actual del servidor FTP. Es interesante advertir que los límites entre la resolución de nombres y el ruteo de mensajes comienzan a desvanecerse.

En las siguientes secciones consideraremos tres clases diferentes de sistemas de nombres. Primero, daremos un vistazo a la manera en que los identificadores se pueden resolver para las direcciones. En este caso, veremos también un ejemplo en donde no es posible distinguir la resolución de nombres del ruteo de mensajes. Después de eso, consideraremos los nombres amigables para el usuario y los nombres descriptivos (es decir, entidades que se describen mediante una colección de nombres).

5.2 NOMBRES PLANOS

Líneas arriba explicamos que los identificadores son convenientes para representar a las entidades de manera única. En muchos casos, los identificadores son simplemente cadenas aleatorias de bits, a las cuales nos referimos convenientemente como nombres no estructurados o planos. Una propiedad

importante de tales nombres es que no contienen información alguna con respecto a la ubicación del punto de acceso de su entidad asociada. A continuación, daremos un vistazo a la forma en que se pueden resolver los nombres, o, de manera equivalente, cómo podemos localizar una entidad cuando solamente proporcionamos su identificador.

5.2.1 Soluciones simples

Consideremos primero dos soluciones sencillas para localizar una entidad. Ambas soluciones son aplicables solamente para redes de área local. Sin embargo, en dicho ambiente, por lo general hacen bien el trabajo, y vuelven particularmente atractiva su simplicidad.

Transmisión y multitransmisión

Considere un sistema distribuido construido sobre una red de computadoras que ofrece capacidades eficientes para la transmisión. Por lo general, dichas capacidades se ofrecen mediante redes de área local en las cuales todas las máquinas están conectadas a un solo cable o a su equivalente lógico. Además, las redes inalámbricas de área local entran en esta categoría.

Localizar una entidad en dicho ambiente es sencillo: un mensaje que contiene al identificador de la entidad es transmitido a cada máquina y a cada máquina se le solicita verificar si tiene una entidad. Solamente las máquinas que pueden ofrecer un punto de acceso para la entidad envían una respuesta que contiene la dirección de dicho punto de acceso.

Este principio se utiliza en el **Protocolo de Resolución de Direcciones** (**ARP**, por sus siglas en inglés) en internet para encontrar la dirección de vínculo de datos de una máquina cuando solamente se tiene la dirección IP (Plummer, 1982). En esencia, una máquina difunde un paquete en la red local que pregunta quien es el propietario de una dirección IP dada. Cuando el mensaje llega a la máquina, el receptor verifica si debiera poner atención a la petición de dirección IP. Si es así, envía un paquete de respuesta que contiene, por ejemplo, su dirección ethernet.

La transmisión se torna ineficiente al crecer la red. No solamente se desperdicia el ancho de banda en los mensajes de petición, más aún, y más seriamente, demasiados servidores pueden ser interrumpidos por peticiones que no pueden responder. Una posible solución es implementar una multitransmisión mediante la cual solamente un grupo restringido de servidores reciba la petición. Por ejemplo, las redes ethernet soportan multitransmisión al nivel de enlace de datos de manera directa con el hardware.

También es posible utilizar la multidifusión para localizar entidades dentro de redes punto a punto. Por ejemplo, internet soporta multitransmisión a nivel de red al permitir a los servidores conectarse a un grupo específico de multitransmisión. Dichos grupos se identifican mediante una dirección de multitransmisión. Cuando un servidor envía un mensaje a una dirección de multitransmisión, la capa de red proporciona un servicio del mejor esfuerzo para enviar dicho mensaje a todos los miembros del grupo. En Deering y Cheriton (1990) y Deering y colaboradores (1996) podrá encontrar información relacionada con implementaciones eficientes para multitransmisión en internet.

Podemos utilizar una dirección de multitransmisión como un servicio general de localización para múltiples entidades. Por ejemplo, considere una empresa en donde cada empleado tiene su

propia computadora móvil. Cuando tal computadora se conecta a una red local disponible, se le asigna una dirección IP de manera automática. Además, se conecta a un grupo específico de multitransmisión. Cuando un proceso desea localizar a la computadora *A*, envía una petición "¿En dónde está *A*?" al grupo de multitransmisión. Si *A* está conectada, responde con su dirección IP actual.

Otra manera de utilizar la dirección de multitransmisión es asociarla con una entidad replicada y utilizar la multitransmisión para localizar la réplica más *cercana*. Al enviar una petición a la dirección de multitransmisión, cada réplica responde con su propia dirección IP actual (normal). Una manera ruda de seleccionar la réplica más cercana es elegir aquella cuya respuesta llegue antes. Explicaremos las demás formas de selección en capítulos posteriores. Como podemos ver, la selección de la réplica más cercana por lo general no es tan fácil.

Apuntadores hacia adelante

Otro método popular para localizar las entidades móviles es utilizar los apuntadores hacia adelante (Fowler, 1985). El principio es simple: cuando una entidad se mueve de *A* a *B*, deja en *A* una referencia de su nueva ubicación en *B*. La principal ventaja de este método es su simplicidad: tan pronto como una entidad es localizada, por ejemplo, mediante el uso del servicio de nombres tradicional, un cliente puede buscar la dirección actual siguiendo la cadena de apuntadores hacia adelante.

También existe cierto número de desventajas. Primero, si no tomamos medidas especiales, para una entidad altamente móvil la cadena que lo localiza puede tornarse demasiado larga y prohibitivamente costosa. Segundo, todas las ubicaciones intermedias localizadas en una cadena tendrán que mantener su parte de la cadena de apuntadores hacia adelante mientras sea necesario. Una tercera (y relacionada) desventaja es la vulnerabilidad de los vínculos rotos. Tan pronto como se pierde un apuntador (por cualquier razón), no se puede llegar a la entidad. Por tanto, una cuestión importante es mantener las cadenas relativamente cortas, y asegurarse de que los apuntadores hacia adelante estén fortalecidos.

Para comprender mejor cómo funcionan los apuntadores hacia adelante, consideremos su uso con respecto a los objetos remotos: objetos a los que se puede acceder por medio de una llamada a un procedimiento remoto. Si seguimos el método de **cadenas SSP** (Shapiro y cols., 1992), cada apuntador hacia adelante se implementa como un par (*resguardo del cliente*, *resguardo del servidor*), según ilustra la figura 5-1. (Vemos que en la terminología original de Shapiro, un resguardo del servidor se conoce como scion y produce pares (*resguardo*, *scion*), lo cual explica su nombre.) Un resguardo del servidor contiene ya sea una referencia local como el objeto real o la referencia local hacia un resguardo remoto del cliente para dicho objeto.

Cada vez que un objeto se mueve desde el espacio de dirección *A* hacia *B*, deja atrás el resguardo del cliente en su lugar en *A* y coloca un resguardo del servidor que hace referencia a él en *B*. Un aspecto interesante de este método es que la migración resulta completamente transparente para el cliente. Lo único que el cliente ve de un objeto es el resguardo del cliente. Cómo, y hacia cuál ubicación envía dicho resguardo del cliente sus invocaciones, permanece oculto para el cliente. Además, observemos que este uso de los apuntadores hacia adelante no es como buscar una dirección. En vez de eso, una petición del cliente se reenvía junto con la cadena real del objeto.

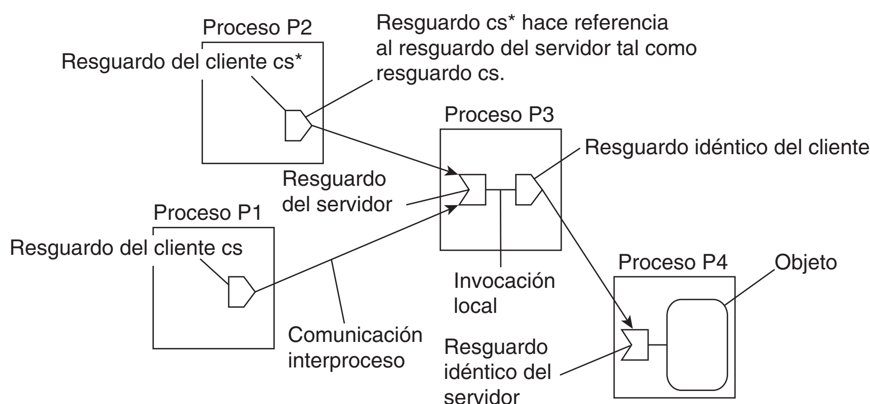


Figura 5-1. El principio de los apuntadores hacia adelante mediante el uso de (*resguardo del cliente, resguardo del servidor*).

Para acortar la cadena de pares (*resguardo del cliente, resguardo del servidor*), la invocación de un objeto acarrea la identificación del resguardo del cliente desde donde se inició dicha invocación. Una identificación del resguardo del cliente consta de la dirección a nivel de transporte del cliente, combinada con un número generado de manera local para identificar dicho resguardo. Cuando la invocación alcanza al objeto en su ubicación actual, se envía una respuesta de vuelta hacia el resguardo del cliente en donde se inició la invocación (con frecuencia sin regresar a la cadena). La ubicación local es preparada junto con esta respuesta, y el resguardo del cliente ajusta su resguardo de servidor acompañante al que se encuentra en la ubicación actual del objeto. Podemos ver este principio en la figura 5-2.

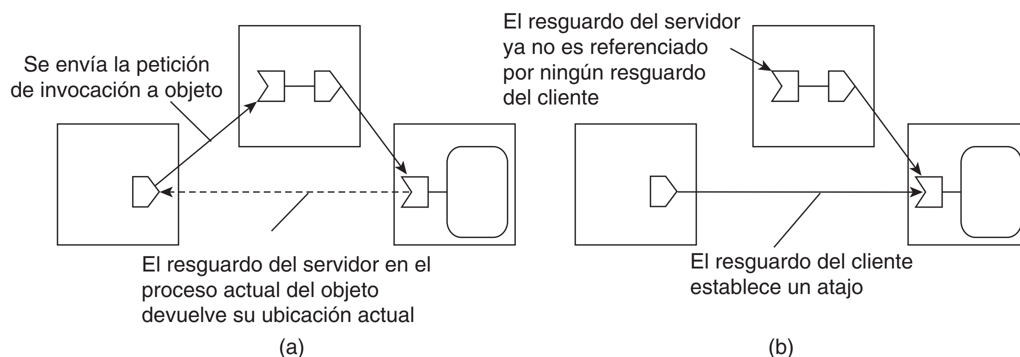


Figura 5-2. Redireccionamiento de un apuntador hacia adelante mediante el almacenamiento de un atajo en el resguardo del cliente.

Existe un intercambio entre el envío de una respuesta de manera directa a la inicialización del resguardo del cliente, o lo largo de la ruta inversa de los apuntadores hacia adelante. En el caso anterior, la comunicación es más rápida debido a que se necesita pasar por menos procesos. Por otro

lado, solamente se puede ajustar el resguardo inicial del cliente, mientras que el envío de la respuesta junto con la ruta inversa permite un ajuste de todos los resguardos intermedios.

Cuando un resguardo del servidor ya no es referido por cliente alguno, es posible eliminarlo. Por sí mismo, esto está muy relacionado con el recolector distribuido de basura, un problema lejos de lo trivial que por lo pronto no veremos aquí. Al lector interesado le recomendamos consultar Abdullahi y Ringwood (1998), Plainfosse y Shapiro (1995), y Veiga y Ferreira (2005).

Ahora suponga que el proceso P_1 de la figura 5-1 pasa su referencia al objeto O para efectuar el proceso P_2. El paso por referencia se lleva a cabo mediante la instalación de una copia p' del resguardo de cliente p en el espacio de dirección del proceso P_2. El resguardo de cliente p' hace referencia al mismo resguardo del servidor que p, de modo que el mecanismo de invocación hacia adelante trabaja igual que antes.

Los problemas surgen cuando un proceso en la cadena de pares (*resguardo del cliente*, *resguardo del servidor*) se arruina o, de algún modo, se vuelve inalcanzable. Muchas soluciones son posibles. Una posibilidad, como la seguida en Emerald (Jul y cols., 1988) y en el sistema LII (Black y Artsy, 1990), es dejar que la máquina en donde se creó el objeto (llamada **ubicación de origen** del objeto) mantenga siempre una referencia hacia su ubicación actual. Esta referencia se almacena y mantiene en una forma de tolerancia a fallas. Cuando se rompe una cadena, se le pregunta a la ubicación de origen del objeto en dónde se encuentra entonces el objeto. Para permitir que la ubicación de origen de un objeto cambie, es posible utilizar un servicio de nombres tradicional para registrar la ubicación de origen actual. A continuación explicaremos estos métodos basados en el origen.

5.2.2 Métodos basados en el origen

El uso de transmisión en los apuntadores hacia adelante establece problemas de escalabilidad. La transmisión y la multitransmisión son difíciles de implementar en redes a gran escala en donde grandes cadenas de apuntadores hacia adelante generan problemas de rendimiento y son susceptibles al rompimiento de los vínculos.

Un método popular usado para soportar entidades móviles dentro de redes a gran escala es introducir la **ubicación de origen**, la cual mantiene el registro de la ubicación actual de una entidad. Se pueden emplear técnicas especiales para salvaguarda por fallas de proceso o de red. En la práctica, con frecuencia se elige la ubicación de origen para que sea el lugar donde la entidad fue creada.

El método basado en el origen se utiliza como un mecanismo para servicios de ubicación con respecto a los apuntadores hacia adelante, según explicamos anteriormente. Otro ejemplo en donde se sigue el método basado en el origen es en las IP móviles (Johnson y cols., 2004), el cual explicamos brevemente en el capítulo 3. Cada servidor móvil utiliza una dirección fija de IP. Toda la comunicación con dicha dirección IP se direcciona inicialmente a los servidores móviles del **agente de origen**. Este agente de origen se localiza en la red de área global que corresponde a la dirección de red contenida en la dirección IP de los servidores móviles. En el caso de IPv6, se representa como un componente de la capa de red. Cada vez que el servidor móvil se mueve hacia otra red, solicita una dirección temporal que podemos utilizar para comunicación. Esta **dirección añadida cuidadosamente** se registra en el agente de origen.

Cuando el agente de origen recibe un paquete para el servidor móvil, busca la ubicación actual del cliente. Si el servidor se encuentra en la red local, el paquete simplemente se reenvía. De lo contrario, se entuba en la ubicación actual del cliente, esto es, se enmascara como dato en un paquete IP y se envía a la dirección añadida cuidadosamente. Al mismo tiempo, al emisor del paquete se le informa de la ubicación del servidor. Este principio aparece en la figura 5-3. Observe que la dirección IP se utiliza de manera efectiva como un identificador para el servidor móvil.

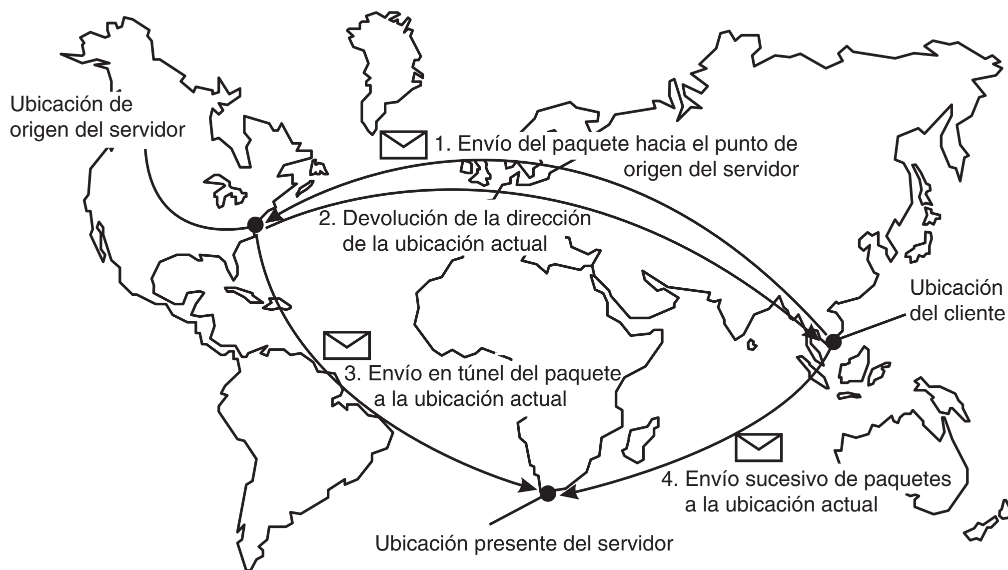


Figura 5-3. El principio de la IP móvil.

La figura 5-3 también ilustra otra desventaja de los métodos basados en el origen en redes de gran escala. Para comunicarse con una entidad móvil, un cliente debe hacer primero contacto con el origen, el cual pudiera estar en una ubicación completamente diferente a la propia entidad. El resultado es un incremento en la latencia de comunicación.

Una desventaja del método basado en el origen, es el uso de una ubicación de origen fija. Por alguna razón, se debe asegurar que la ubicación de origen exista siempre. De lo contrario, será imposible contactar a la entidad. Los problemas se agravan cuando una entidad de vida larga decide moverse de manera permanente hacia una parte completamente diferente de la red que donde está localizado su origen. En ese caso, sería mejor que el origen se mueva junto con el servidor.

Una solución a este problema es registrar el origen de un servicio de nombres tradicional y permitir a un cliente buscar primero la ubicación del origen. Debido a que podemos asumir la ubicación de origen como relativamente estable, se le puede aplicar un caché después de haberla buscado.

5.2.3 Tablas Hash distribuidas

Ahora demos un vistazo a desarrollos recientes para resolver un identificador para la dirección de la entidad asociada. Ya mencionamos en varias ocasiones las tablas Hash distribuidas, pero hemos diferido la explicación sobre la manera en que realmente trabajan. En este apartado corregiremos esta situación considerando primero el sistema de cuerdas como un sencillo sistema DHT fácil de explicar. En su forma más simple, los sistemas basados en DHT no consideran la proximidad de redes en lo absoluto. Esta negligencia pudiera provocar problemas de rendimiento. También explicaremos las soluciones apropiadas para sistemas predispuestos para una red.

Funcionamiento general

Existen distintos sistemas basados en DHT, acerca de los cuales aparece una breve explicación en Balakrishnan y colaboradores (2003). El sistema de cuerdas (Stoica y cols., 2003) representa a muchos sistemas DHT, aunque existen diferencias sutiles pero importantes que influencian su complejidad para los protocolos de administración y búsqueda. Como explicamos brevemente en el capítulo 2. El sistema de cuerdas utiliza un identificador de m bits de espacios para asignar los identificadores de manera aleatoria hacia los nodos, así como las llaves para especificar las entidades. Estas últimas pueden ser virtualmente cualquier cosa: archivos, procesos, etc. El número m de bits por lo general es de 128 o 160, dependiendo de cuál función hash se utilice. Una entidad con llave k entra en la jurisdicción del nodo que tiene el identificador más pequeño $id \geq k$. A este nodo lo conocemos como el *sucesor* de k, y se representa como $succ(k)$.

En los sistemas basados en DHT, la cuestión principal es resolver de manera eficiente una llave k para la dirección de $succ(k)$. Un método evidente no escalable es dejar que cada nodo p mantenga el registro del sucesor $succ(p + 1)$ así como el de su predecesor $pred(p)$. En ese caso, cada vez que un nodo p reciba una petición para resolver la llave k, simplemente reenviará la petición a uno de sus vecinos (el que resulte apropiado) a menos que $pred(p) < k \leq p$, en cuyo caso el nodo p debe devolver su propia dirección hacia el proceso que inició la resolución de la llave k.

En lugar de este método lineal basado en la búsqueda de la llave, cada nodo de cuerdas mantiene una **tabla finger** para la mayoría de las entradas m. Si FT_p denota la tabla finger del nodo p, entonces

$$FTp[i] = succ(p + 2^{i-1})$$

Puesto en otras palabras, los i-ésimos puntos de entrada al primer nodo suceden a p en por lo menos 2^{i-1}. Observemos que estas referencias son en realidad un atajo hacia los nodos existentes en el espacio del identificador i, en donde la distancia reducida desde el nodo p se incrementa de manera exponencial conforme se incrementa el índice en la tabla finger. Para efectuar una búsqueda de la llave k, el nodo p reenviará entonces la petición de inmediato al nodo q cuyo índice es j en la tabla finger de p, en donde:

$$q = FT_p[j] \leq k < FT_p[j + 1]$$

(Para mayor claridad, ignoraremos el módulo aritmético.)

Para ilustrar esta búsqueda, solucione el caso para $k = 26$ del nodo 1 como aparece en la figura 5-4. Primero, el nodo 1 buscará $k = 26$ dentro de su tabla finger para descubrir que este valor es mayor que $FT_1[5]$, lo cual significa que la petición se envía al nodo 18 $= FT_1[5]$. El nodo 18, a su vez, seleccionará al nodo 20, conforme $FT_{18}[2] < k \leq FT_{18}[3]$. Por último, la petición es enviada desde el nodo 20 al nodo 21 y de ahí al nodo 28, el cual es responsable de $k = 26$. En ese punto, la dirección del nodo 28 se devuelve al nodo 1 y la llave se resuelve. Por razones similares, cuando se le pide al nodo 28 que resuelva la llave $k = 12$, la petición será enrutada como podemos ver en la línea discontinua de la figura 5-4. Podemos mostrar que la búsqueda requiere, por lo general, O(*log*(*N*)) pasos, donde *N* es el número de nodos presentes en el sistema.

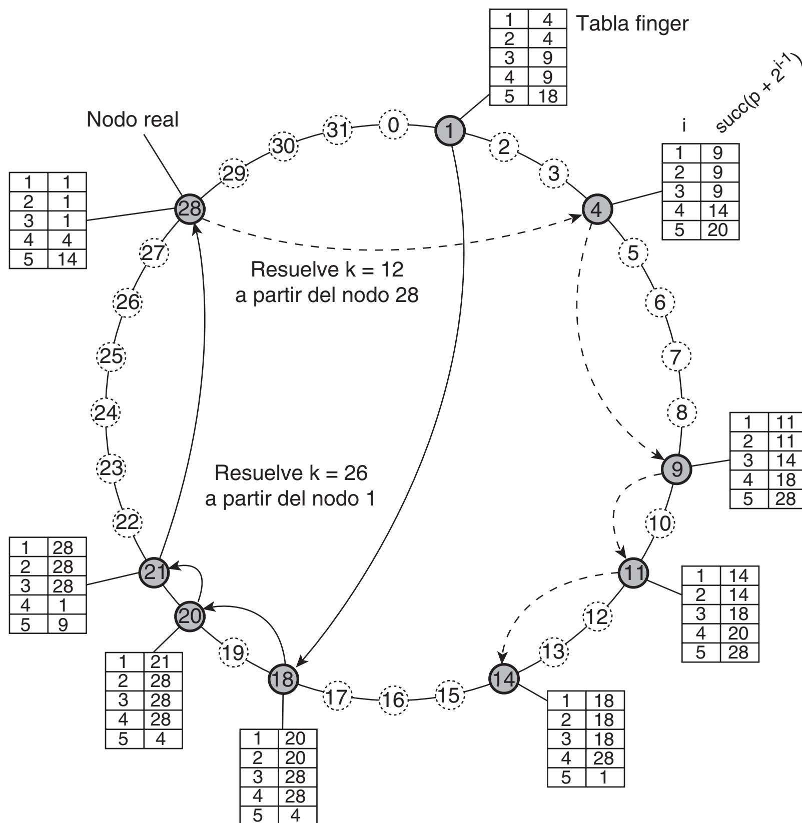


Figura 5-4. Resolución de la llave 26 del nodo 1 y la llave 12 del nodo 28 en el sistema de cuerdas.

En sistemas distribuidos grandes podemos esperar que la colección de nodos participantes cambie todo el tiempo. No solamente los nodos entrarán y saldrán de manera voluntaria, necesitamos también considerar el caso de nodos que fallan (y de esta manera abandonan efectivamente el sistema), para luego recuperarse nuevamente (en cuyo punto vuelven a entrar).

Unirse a un sistema basado en DHT es relativamente simple, tal como en el sistema de cuerdas. Suponga que el nodo p desea unirse. Simplemente hace contacto con un nodo arbitrario en el sistema existente y solicita una búsqueda para $succ(p + 1)$. Una vez identificado este nodo, p puede insertarse a sí misma dentro del anillo. De manera similar, abandonar puede ser igual de simple. Observe que los nodos también mantienen un registro de su predecesor.

Desde luego, la complejidad proviene de mantener las tablas finger actualizadas. Más importante es que para cada nodo q, $FT_q[1]$ es correcto dado que esta entrada hace referencia al siguiente nodo presente en el anillo, esto es, el sucesor de $q + 1$. Con el objeto de lograr esta meta, por lo general, cada nodo q ejecuta un sencillo procedimiento que hace contacto con $succ(q + 1)$ y hace la petición de devolver $pred(succ(q + 1))$. Si $q = pred(succ(q + 1))$, entonces q sabe que su información es consistente con la de su sucesor. De lo contrario, si el sucesor de q actualiza a su predecesor, entonces, de manera aparente, un nuevo nodo p entra al sistema, con $q < p \leq succ(q + 1)$, de modo que q ajustará $FT_q[1]$ para p. En ese punto, también verificará si p registra a q como su predecesor. Si no, se requiere otro ajuste de $FT_q[1]$.

De manera similar, para actualizar una tabla finger, el nodo q solamente necesita encontrar al sucesor de $k = q + 2^{i+1}$ para cada entrada i. De nuevo, esto se puede hacer al lanzar una petición para resolver $succ(k)$. En el sistema de cuerdas, dichas peticiones se lanzan de manera regular por medio de un proceso localizado en segundo plano.

De igual modo, cada nodo q verificará regularmente si su predecesor está vivo. Si el predecesor falla, la única cosa que puede registrar q es el hecho de configurar $pred(q)$ a “desconocido”. Por otro lado, cuando el nodo q actualice su vínculo al siguiente nodo conocido en el anillo, y se dé cuenta de que el predecesor de $succ(q + 1)$ se estableció como “desconocido”, simplemente notificará a $succ(q + 1)$ que sospecha que es el predecesor. Por tanto, estos sencillos procedimientos aseguran que un sistema de cuerdas es por lo general consistente, con excepción quizá de sólo unos cuantos nodos. Podemos encontrar los detalles en Stoica y colaboradores (2003).

Exploración de la proximidad en red

Uno de los potenciales problemas con sistemas tales como el de cuerdas es que las peticiones se pueden rutear de manera errática por toda la internet. Por ejemplo, asuma que el nodo 1 de la figura 5-4 se ubica en Amsterdam, Holanda; el nodo 18 en San Diego, California; el nodo 20 de nuevo en Amsterdam; y el nodo 21 en San Diego. Entonces, el resultado de resolver la llave 26 incurrirá en la transferencia de tres mensajes de área amplia que podríamos argumentar se redujeron cuando mucho a uno. Para minimizar estos casos patológicos, el diseño de un sistema basado en DHT requiere tomar en cuenta la red subyacente.

Castro y colaboradores (2002b) distinguen tres diferentes formas de hacer un sistema basado en DHT consciente de la red subyacente. En el caso de la **asignación de nodos basados en la topología**, la idea es asignar identificadores de tal modo que dos nodos próximos tengan identificadores que también se encuentren cercanos entre sí. No es difícil imaginar que este método puede imponer problemas severos en el caso de sistemas relativamente sencillos tales como el de cuerdas. Cuando los identificadores se ejemplifican desde un espacio unidimensional, mapear un anillo lógico en internet está lejos de lo trivial. Más aún, dicho mapeo puede fácilmente exponer fallas correlacionadas: los nodos presentes en la misma red corporativa tendrán identificadores a partir de un intervalo

relativamente pequeño. Cuando esa red se vuelve inalcanzable, tenemos repentinamente un hueco en la distribución alternativa uniforme de identificadores.

Con el **ruteo por proximidad**, los nodos mantienen una lista de alternativas para reenviar una petición. Por ejemplo, en lugar de tener solamente un sucesor, cada nodo en el sistema de cuerdas pudiera de igual manera mantener el registro de los sucesores de r. De hecho, esta redundancia se puede aplicar para cada entrada dentro de una tabla finger. Para el nodo p, los puntos $FT_p[i]$ hacia el primer nodo en el rango $[p + 2^{i-1}, p + 2^i - 1]$. No existe razón alguna por la que p no pueda mantener un registro de los nodos r en dicho rango: si es necesario, cada uno de tales nodos se puede utilizar para rutear una petición de búsqueda para una llave $k > p + 2^i - 1$. En ese caso, al elegir el reenvío de una petición de búsqueda, un nodo puede recolectar uno de los sucesores de r que se encuentre cerca de sí mismo, pero además satisface la restricción de que el identificador del nodo elegido debe ser más pequeño que el propuesto por la llave solicitada. Una ventaja adicional de tener sucesores múltiples para cada entrada de la tabla es que las fallas presentes en los nodos no necesariamente necesitan llevar las fallas de las búsquedas, como rutas múltiples que se pueden explorar.

Por último, en la **selección de proximidad del vecino** la idea es optimizar las tablas de ruteo en tal forma que el nodo más cercano se seleccione como vecino. Esta selección trabaja solamente cuando existen más nodos de dónde elegir. En un sistema cuerdas, por lo general, no es el caso. Sin embargo, en otros protocolos tales como Pastry (Rowstron y Druschel, 2001), cuando un nodo se integra recibe la información con respecto al recubrimiento actual desde otros nodos múltiples. Esta información se utiliza mediante un nuevo nodo para construir una tabla de ruteo. Desde luego, cuando existen nodos alternativos entre los cuales elegir, la selección de proximidad del vecino permitirá que el nodo elegido para integrar sea el mejor.

Observemos que pudiera no ser tan fácil trazar una línea entre la proximidad de ruteo y la selección de proximidad del vecino. De hecho, cuando se modifica el sistema cuerdas para incluir sucesores de r para cada entrada de la tabla finger, se recurre a la selección de proximidad del vecino para identificar a los vecinos r más cercanos, los cuales están muy cerca del ruteo por proximidad como ya explicamos (Dabek y cols., 2004b).

Para terminar, vemos también que se puede hacer una distinción entre **iterativo** y **búsqueda recursiva**. En el caso anterior, un nodo al cual se solicita la búsqueda de una llave devolverá la dirección de red del siguiente nodo encontrado para el proceso que hace la petición. Entonces, el proceso solicitará que el siguiente nodo lleve a cabo otro paso para resolver la llave. Una alternativa, y esencialmente la forma que ya explicamos, es permitir que un nodo reenvíe una petición de búsqueda al siguiente nodo. Ambos métodos tienen sus ventajas y desventajas, las cuales explicaremos más adelante en este capítulo.

5.2.4 Métodos jerárquicos

En esta sección explicaremos primero un método general para trazar un esquema de ubicación jerárquica, después del cual presentaremos un número de optimizaciones. El método que presentamos está basado en el servicio de localización Globe, descrito con detalles en Ballintijn (2003). Podemos encontrar una revisión en Van Steen y colaboradores (1998b). Éste es un servicio general de

ubicación representativo de muchos servicios de ubicación jerárquicos propuestos para lo que denominamos sistemas de comunicación personal, de los cuales podemos encontrar una revisión general en Pitoura y Samaras (2001).

En un sistema jerárquico, una red está dividida en una colección de **dominios**. Existe un solo dominio de nivel superior que se expande por toda la red. Cada dominio se puede subdividir en múltiples subdominios más pequeños. Un dominio de nivel muy bajo, llamado **dominio hoja**, por lo general corresponde a una red de área local ubicada en una red de computadoras o una célula de red de telefonía móvil.

Cada dominio *D* tiene un nodo de directorio asociado *dir(D)* que mantiene el registro de las entidades en cada dominio. Esto genera un árbol de nodos de directorio. El nodo de directorio del dominio de nivel superior, llamado **nodo raíz (directorio)**, sabe acerca de todas las entidades. Esta organización general de una red dentro de dominios y nodos de directorio se muestra en la figura 5-5.

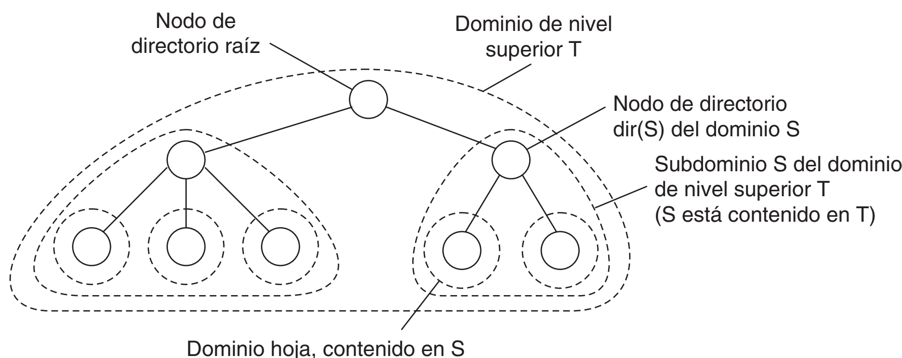


Figura 5-5. Organización jerárquica de un servicio de ubicación dentro de dominios, cada dominio tiene asociado un nodo de directorio.

Para mantener un registro con respecto a los pormenores de una entidad, cada entidad en curso está localizada dentro de un dominio *D* representado por un **directorio de ubicación** dentro del nodo directorio *dir(D)*. Un registro de localización para una entidad *E* en el nodo de directorio *N* para un dominio *D* contiene la dirección actual de una entidad dentro de dicho dominio. Por el contrario, el nodo directorio *N′* para el siguiente dominio de más alto nivel *D′* que contiene a *D*, tendrá el registro de localización para *E* que contiene solamente un apuntador a *N*. De manera similar, el nodo padre de *N′* almacenará un registro de localización para *E* que contiene sólo un apuntador a *N′*. Por consecuencia, el nodo raíz tendrá un registro de localización para cada entidad, en donde cada registro de localización almacena un apuntador al nodo de directorio del siguiente subdominio de más bajo nivel en que se ubica actualmente la entidad asociada al registro.

Una entidad puede contener múltiples direcciones, por ejemplo, si se replica. Si una entidad tiene una dirección y un dominio hoja D_1 y D_2, respectivamente, entonces el nodo de directorio del dominio más pequeño contiene tanto a D_1 como a D_2, tendremos dos apuntadores, uno por cada

subdominio que contiene una dirección. Esto produce la organización general del árbol tal como vemos en la figura 5-6.

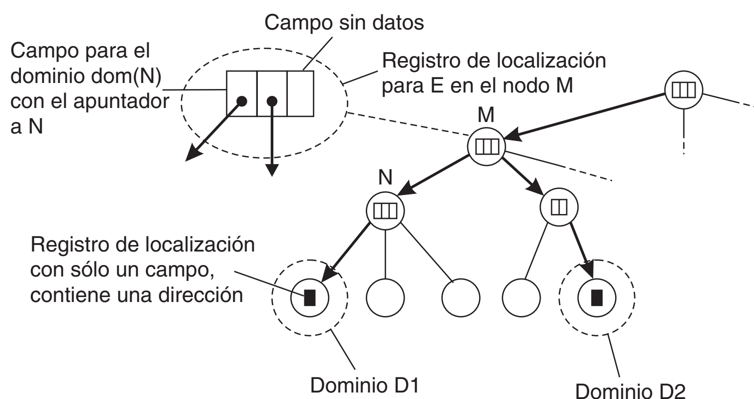


Figura 5-6. Ejemplo de almacenamiento de información de una entidad que tiene dos direcciones en diferentes dominios hoja.

Consideremos ahora la manera en que una operación de búsqueda se ejecuta en dicho servicio de ubicación. Como podemos ver en la figura 5-7, un cliente que desea localizar una entidad *E* lanza una petición de búsqueda al nodo de directorio localizado en el dominio hoja *D* en el cual reside el cliente. Si el nodo de directorio no almacena un registro de ubicación para la entidad, entonces la entidad no se localiza en *D*. En consecuencia, el nodo reenvía la petición a su padre. Observe que el nodo padre representa un dominio más grade que su hijo. Si el padre tampoco tiene un registro en *E*, la petición de búsqueda se reenvía al siguiente nivel más alto, y así sucesivamente.

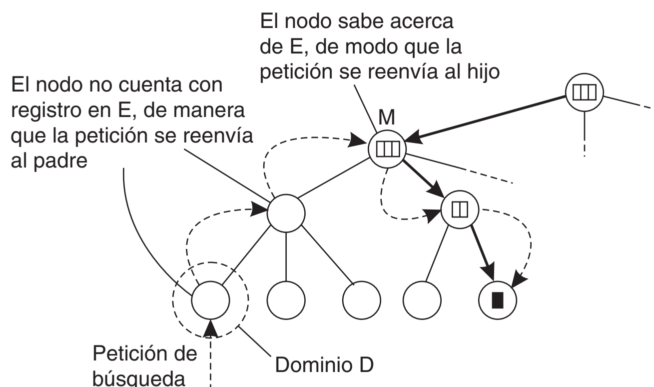


Figura 5-7. Búsqueda de una ubicación dentro de un servicio de ubicación organizado de manera jerárquica.

Tan pronto como la petición alcanza un nodo de directorio *M* que almacena el registro de ubicación para la entidad *E*, sabemos que *E* se encuentra en alguna parte del dominio *dom(M)* repre-

sentado por el nodo *M*. En la figura 5-7, *M* se muestra para almacenar el registro de la ubicación que contiene un apuntador a uno de sus subdominios. Entonces, la petición de búsqueda se reenvía al nodo de directorio de dicho subdominio, el cual a cambio lo reenvía hacia abajo en el árbol hasta que finalmente la petición alcanza una hoja del árbol. El registro de ubicación almacenado en la hoja del árbol en el nodo hoja contendrá la dirección de *E* en el dominio hoja. Entonces, esta dirección puede devolverse al cliente que inicialmente solicitó la búsqueda que se llevará a cabo.

Una observación importante con respecto a los servicios de ubicación jerárquica es que la operación de búsqueda hace la exploración de manera local. En principio, la búsqueda de la entidad se efectúa incrementando gradualmente el anillo centrado alrededor del cliente sujeto a petición. El área de búsqueda se expande cada vez que la petición se reenvía a la dirección de más alto nivel del nodo de directorio. En el peor de los casos, la búsqueda continúa hasta que la petición alcanza el nodo raíz. Debido a que el nodo raíz contiene un registro de ubicación para cada entidad, entonces la petición puede simplemente reenviarse a lo largo de una ruta descendente de apuntadores hacia uno de los nodos hoja.

Las operaciones de actualización se realizan similarmente de manera local, tal como ilustra la figura 5-8. Considere una entidad *E* que crea una réplica en el dominio hoja *D* para el cual se necesita insertar su dirección. La inserción se inicia en el nodo hoja *dir(D)* de *D* el cual reenvía de inmediato la petición insertada a su padre. El padre reenviará también la petición de inserción, hasta que alcance un nodo de directorio *M* que de antemano almacena un registro de ubicación para *E*.

El nodo *M* almacenará entonces un apuntador en el registro de localización para *E*, haciendo referencia al nodo desde el cual fue reenviada la petición de inserción. En ese punto, el nodo hijo crea un registro de localización para *E* que contiene un apuntador al siguiente nodo de más bajo nivel desde donde proviene la petición. Este proceso continúa hasta que alcancemos el nodo hoja desde donde inició la inserción. El nodo hoja, finalmente, crea un registro con la dirección de la entidad en el dominio hoja asociado.

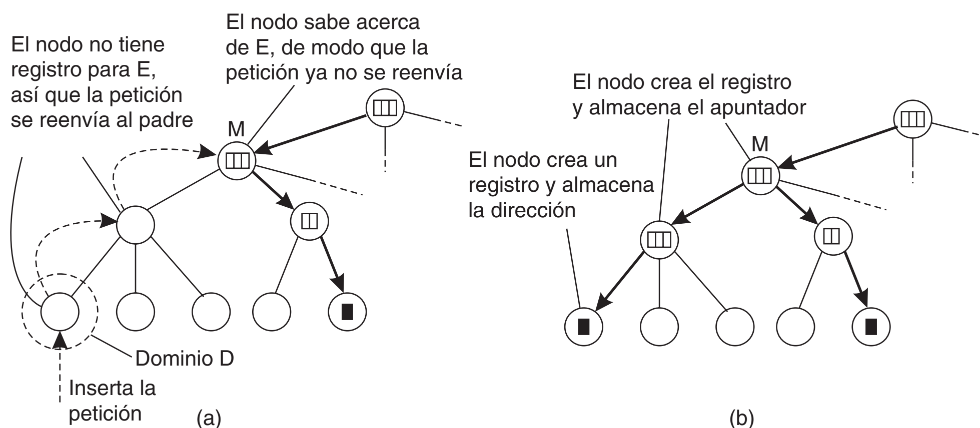


Figura 5-8. (a) Una petición de inserción se reenvía al primer nodo que sabe acerca de la entidad E. (b) Se crea una cadena de apuntadores hacia adelante para el nodo hoja.

Insertar una dirección tal y como se describió permite la instalación de la cadena de apuntadores en una modalidad arriba-abajo comenzando en el nodo directorio de menor nivel que tenga un registro de ubicación para la entidad *E*. Una alternativa es crear un registro de ubicación antes de pasar la petición inserción al nodo padre. En otras palabras, la cadena de apuntadores se construye de abajo hacia arriba. La ventaja de lo anterior es que una dirección queda disponible para búsquedas lo más pronto posible. Por consecuencia, si un nodo padre es temporalmente inalcanzable, la dirección puede buscarse aún dentro del dominio representado por el nodo actual.

Una operación de eliminación es análoga a una operación de inserción. Cuando necesitamos remover una dirección para la entidad *E* en el dominio hoja *D*, se le solicita al nodo de directorio *dir(D)* que remueva dicha dirección desde su registro de ubicación para *E*. Si dicho registro de localización llega a quedarse vacío, es decir, que no contenga ninguna otra dirección para *E* en *D*, se puede remover. En tal caso, el nodo padre de *dir(D)* desea remover su apuntador hacia *dir(D)*. Si registro de localización para *E* en el padre se queda vacío, igualmente se debe remover y el siguiente nodo de directorio de más alto nivel debería ser informado. De nuevo, este proceso continúa hasta que se remueve un apuntador desde un registro de localización que permanece no vacío posteriormente o hasta que se alcanza la raíz.

5.3 NOMBRES ESTRUCTURADOS

Los nombres planos son buenos para las máquinas, pero por lo general no muy convenientes para uso de las personas. Como alternativa, los sistemas de nombres con frecuencia soportan nombres estructurados que están compuestos a partir de nombres sencillos y legibles para las personas. No solamente para los nombres de archivo, también en internet los servidores de nombres siguen este método. En esta sección nos concentramos en los nombres estructurados y en la forma en que se resuelven para las direcciones.

5.3.1 Espacios de nombre

Por lo general, los nombres están organizados en lo que conocemos como **espacio de nombre**. Los espacios de nombre se pueden representar como un gráfico etiquetado y dirigido con dos tipos de nodos. Un **nodo hoja** representa una entidad con nombre y tiene la propiedad de que no contiene aristas salientes. Un nodo hoja almacena, generalmente, información con respecto a la entidad que representa —por ejemplo, su dirección— de modo que el cliente puede acceder al nodo. De manera alternativa, puede almacenar el estado de dicha entidad, tal como en el caso de sistemas de archivos donde el nodo hoja realmente contiene el archivo completo al que representa. Más adelante regresaremos al contenido de los nodos.

Al contrario del nodo hoja, el **nodo directorio** contiene cierto número de aristas salientes, cada una etiquetada con un nombre, como podemos ver en la figura 5-9. En un grafo de nombres, cada nodo está considerado sólo como otra entidad del sistema distribuido, y, en especial, como un

identificador asociado. Un nodo directorio almacena una tabla en la cual la arista saliente se representa como un par (*etiqueta de arista, identificador de nodo*). A dicha tabla se le llama **tabla de directorio**.

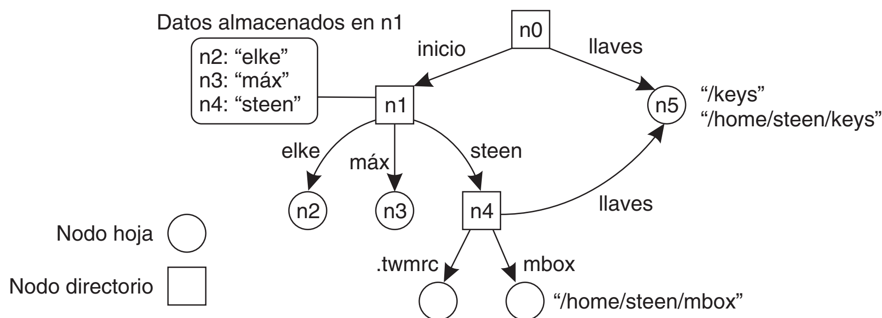


Figura 5-9. Grafo general de nombres con sólo un nodo raíz.

El grafo de nombres que aparece en la figura 5-9 tiene un nodo, a saber n_0, el cual contiene solamente aristas salientes y no entrantes. A tal nodo se le conoce como **nodo (raíz)** del grafo de nombres. Aunque es posible que un grafo tenga varios nodos raíz, por sencillez, muchos sistemas de nombres tienen sólo uno. En un grafo de nombres podemos hacer referencia a cada ruta mediante la secuencia de etiquetas que corresponden a las aristas presentes en dicha ruta, tal como

N:<etiqueta-1, etiqueta-2, ..., etiqueta-n>

donde *N* hace referencia al primer nodo de la ruta. A dicha secuencia se le conoce como **nombre de ruta**. Si el primer nodo presente en el nombre de ruta es la raíz del grafo de nombres, se le llama **nombre de ruta absoluto**. De lo contrario, se le llama **nombre de ruta relativo**.

Es importante darse cuenta de que los nombres siempre se organizan dentro de un espacio de nombre. En consecuencia, un nombre siempre está definido en forma relativa hacia un nodo directorio. En este sentido, el término "nombre absoluto" es, de alguna manera, incorrecto. De manera similar, la diferencia entre nombres locales y globales con frecuencia puede resultar confusa. Un **nombrel global** es un nombre que denota la misma entidad, sin importar en dónde se utilice dentro de un sistema. En otras palabras, un nombre global siempre se interpreta con el mismo nodo directorio. Por el contrario, un **nombre local** es un nombre cuya interpretación depende del lugar en donde se utilice. Dicho de manera diferente, un nombre local es, en esencia, un nombre relativo cuyo directorio en el cual está contenido (implícitamente) se conoce. Regresaremos a estos problemas más tarde, cuando expliquemos la resolución de nombres.

Esta descripción de un grafo de nombres viene al caso con respecto a lo que está implementado en muchos sistemas de archivos. Sin embargo, en lugar de escribir la secuencia de las etiquetas de los extremos para representar el nombre de la ruta, en los sistemas de archivos, por lo general, los nombres de ruta están representados como una sola cadena en la cual las etiquetas están separadas mediante un carácter separador especial, tal como una diagonal ("/"). Este carácter también se

utiliza para indicar si un nombre de ruta es absoluto. Por ejemplo, en la figura 5-9, en lugar de utilizar n_0:<*inicio, steen, mbox*>, esto es, el nombre de ruta real, es práctica común utilizar una representación de cadena */home/steen/mbox*. Además, observe que cuando existen distintas rutas que llegan al mismo nodo, éste se puede representar mediante diferentes nombres de ruta. Por ejemplo, en la figura 5-9 se puede hacer referencia al nodo n_5 mediante */home/steen/keys* igual que utilizando */keys*. La representación de cadena de una ruta de nombres se puede aplicar igualmente a otros grafos de nombres además de a aquellas utilizadas únicamente para los sistemas de archivos. En Plan 9 (Pike y cols., 1995), todos los recursos, tales como procesos, servidores, servicios de E/S, e interfaces de red, son nombrados de igual manera que los archivos tradicionales. Este método es análogo a la implementación de un sol grafo de nombres para todos los recursos presentes en un sistema distribuido.

Existen diferentes maneras de organizar el mismo espacio. Como ya mencionamos, la mayoría de los espacios de nombre tiene solamente un nodo raíz. En muchos casos, un espacio de nombres es además estrictamente jerárquico en el sentido de que el grafo de nombres se organiza como un árbol. Esto significa que cada nodo excepto el nodo raíz tiene exactamente una arista entrante; la raíz no contiene aristas entrantes. En consecuencia, cada nodo tiene también un nombre de ruta (absoluto) asociado.

El grafo de nombres que aparece en la figura 5-9 ejemplifica un *grafo dirigido no cíclico*. En tal organización, un nodo puede tener más de una arista entrante, pero no se le permite al grafo tener un ciclo. Existen también espacios de nombre que no tienen esta restricción.

Para hacerlo más concreto, considere la forma en que se nombran los archivos dentro de un sistema de archivos UNIX tradicional. En el grafo de nombres para UNIX, un nodo directorio representa un directorio de archivos, en donde un nodo hoja representa a un archivo. Existe un solo directorio raíz, representado en el grafo de nombres por el nodo raíz. La implementación del grafo de nombres es parte integral de toda la implementación del sistema de archivos. Dicha implementación consta de series contiguas de bloques a partir de un disco lógico, particionado por lo general en un bloque de inicio (*boot block*), un superbloque, una serie de nodos índice (llamados inodos), y bloques de archivos de datos. Vea también Crowley (1997), Silberschatz y colaboradores (2005), y Tanenbaum y Woodhull (2006). Esta organización aparece en la figura 5-10.

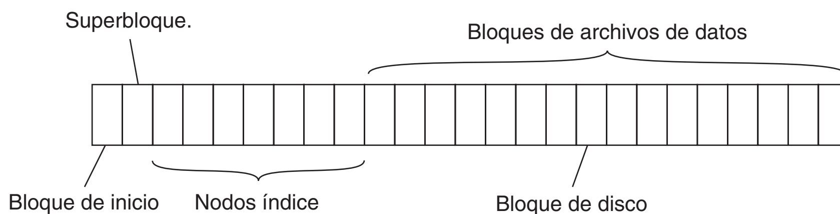


Figura 5-10. Organización general de una implementación del sistema de archivos UNIX en un disco lógico de bloques contiguos de disco.

El bloque de inicio es un bloque especial de datos e instrucciones que se cargan de manera automática dentro de la memoria principal cuando se reinicia el sistema. El bloque de inicio se utiliza para cargar al sistema operativo dentro de la memoria principal.

El superbloque contiene información con respecto a todo el sistema de archivos, tal como su tamaño, qué bloques del disco aún no están ocupados, cuáles inodos aún no fueron utilizados, entre otras cosas más. A los inodos se les referencia mediante un número de índice, que comienza con el número cero, el cual es reservado para el inodo que representa al directorio raíz.

Cada inodo contiene información acerca del lugar en donde se pueden encontrar los datos de su archivo asociado. Además, un inodo contiene información relativa a su propietario, la fecha de creación, y la última modificación, protección, entre otras cosas. En consecuencia, cuando el número de índice está dado por un inodo, es posible acceder a su archivo asociado. Cada directorio se implementa además como un archivo. Éste es también el caso para el directorio raíz, el cual contiene un mapeo entre los nombres de archivo y los números de índice de los inodos. De esta manera, vemos que el número de índice de un inodo corresponde a un identificador de nodo en el grafo de nombres.

5.3.2 Resolución de nombres

Los espacios de nombre ofrecen un mecanismo apropiado para almacenar y recuperar información con respecto a las entidades por medio de nombres. De manera más general, dado el nombre de una ruta, debiera ser posible buscar cualquier información almacenada en el nodo referido por dicho nombre. Al proceso de búsqueda de un nombre se le llama **resolución de nombre**.

Para explicar la manera en que funciona la resolución, consideremos un nombre de ruta como *N*:<$etiqueta_1$, $etiqueta_2$,…, $etiqueta_n$>. La resolución de este nombre comienza en el nodo *N* del grafo de nombres, donde el nombre $etiqueta_1$ se busca en la tabla de directorio y el cual devuelve al identificador del nodo al que hace referencia $etiqueta_1$. La resolución continúa entonces en el nodo identificado al buscar el nombre $etiqueta_2$ en su tabla de directorio, y así sucesivamente. Cuando se asume que la ruta nombrada realmente existe, la resolución se detiene en el último nodo referido como $etiqueta_n$, al devolver el contenido de dicho nodo.

Una búsqueda de nombre devuelve el identificador de un nodo a partir del cual continúa el proceso de resolución de nombres. En especial, es necesario acceder a la tabla de directorio de un nodo identificado. Considere de nuevo un grafo de nombres para un sistema de archivos UNIX. Como ya mencionamos, un identificador de nodo se implementa como un número de índice de un inodo. Acceder a una tabla de directorio significa que debe leerse el primer inodo para descubrir en dónde se encuentran los datos reales almacenados en el disco, y luego de manera subsecuente leer los bloques de datos que contiene la tabla de directorio.

Mecanismo de clausura

La resolución de nombres puede tener lugar solamente si sabemos cómo y en dónde comenzar. En nuestro ejemplo, el nodo de inicio está dado, y asumimos que tenemos acceso a su tabla de directorio. Saber cómo y en dónde comenzar la resolución de nombres es conocido, por lo general, como el **mecanismo de clausura**. De manera esencial, un mecanismo de clausura trata con la selección del nodo inicial dentro de un espacio de nombres en el cual empieza la resolución de nombres (Radia, 1989). Lo que en ocasiones vuelve difícil la comprensión de los mecanismos

de clausura es que, necesariamente, son parte implícita y pueden resultar muy diferentes al compartirlos con otros.

Por ejemplo, para un sistema de archivos UNIX, la resolución de nombres en el grafo de nombres hace uso del hecho de que un inodo del directorio raíz es el primer inodo en el disco lógico que representa al sistema de archivos. El desplazamiento real del byte se calcula a partir de los valores presentes en otros campos del superbloque, junto con la información fija en el propio sistema operativo en la organización del superbloque.

Para aclarar este punto, considere la representación de un nombre de archivo tal como */home/steen/mbox*. Para resolver este nombre, es necesario tener ya acceso a la tabla de directorio del nodo raíz del grafo de nombres apropiado. Ya que es un nodo raíz, no se puede buscar por sí mismo a menos que se implemente como un nodo diferente dentro de otro grafo de nombres, digamos *G*. En consecuencia, resolver un nombre de archivo requiere que ya se encuentre implementado algún mecanismo por medio del cual pueda comenzar el proceso de resolución.

Un ejemplo completamente distinto es el uso de la cadena "0031204430784". Mucha gente no sabrá qué hacer con estos números, a menos de que se le indique que la secuencia es un número de teléfono. Esta información es suficiente para comenzar el proceso de resolución, en especial, mediante la marcación del número. Posteriormente, el sistema de telefonía hace el resto.

Como último ejemplo, considere el uso de nombres locales en los sistemas distribuidos. Un caso típico de un nombre local es un ambiente variable. Por ejemplo, en los sistemas UNIX, la variable nombrada *HOME* se utiliza para hacer referencia al directorio de inicio de un usuario. Cada usuario tiene su propia copia de esta variable, la cual se inicializa con el nombre global de sistema que corresponde al directorio de inicio del usuario. El mecanismo de cierre asociado con las variables de ambiente asegura que el nombre de la variable se resuelva de manera apropiada mediante la búsqueda de una tabla específica de usuario.

Vinculación y montaje

El uso de un **alias** se encuentra fuertemente relacionado con la resolución de nombres. Un alias es otro nombre para la misma entidad. Una variable de ambiente es ejemplo de un alias. En términos del grafo de nombres, existen básicamente dos maneras de implementar un alias. El primer método consiste en simplemente permitir múltiples rutas absolutas de nombres para hacer referencia al mismo nodo en el grafo de nombres. Este método aparece en la figura 5-9, en la cual el nodo n_5 puede ser referido mediante dos nombres de ruta diferentes. En la terminología UNIX, los dos nombres de ruta */keys* y */home/steen/keys* de la figura 5-9 se conocen como **vínculos absolutos** hacia el nodo n_5.

El segundo método es representar una entidad mediante un nodo hoja, digamos *N*, pero en lugar de almacenar la dirección o estado de dicha entidad, el nodo almacena un nombre de ruta absoluto. Al resolver primero un nombre de ruta absoluto que guía a *N*, la resolución de nombres devolverá el nombre de ruta almacenado en *N*, en cuyo punto puede continuar con la resolución del nuevo nombre de ruta. En los sistemas de archivos UNIX, este principio corresponde al uso de **vínculos simbólicos**, y lo podemos ver en la figura 5-11. En este ejemplo, el nombre de la ruta */home/steen/keys*, el cual hace referencia al nodo que contiene el nombre de ruta absoluto */keys*, como un vínculo simbólico al nodo n_5.

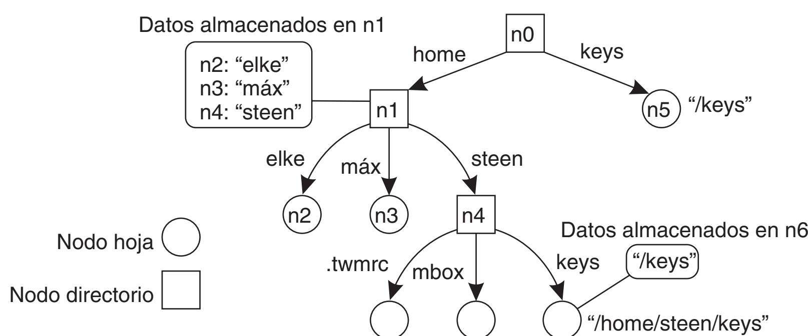


Figura 5-11. Concepto de un vínculo simbólico explicado mediante un grafo de nombres.

La resolución de nombres descrita hasta aquí tiene lugar por completo dentro de un solo espacio de nombres. Sin embargo, la resolución de nombres también se puede utilizar para mezclar diferentes espacios de nombre en forma transparente. Consideremos primero un sistema de archivos montado. En términos de nuestro sistema de nombres, un sistema de archivos montado corresponde a dejar que un nodo directorio almacene el identificador en un nodo directorio desde un espacio de nombre *diferente*, al cual hacemos referencia mediante un espacio de nombre externo. Al nodo directorio que almacena el identificador del nodo se le llama **punto de montaje**. En consecuencia, al nodo directorio localizado en el espacio de nombre externo se le llama **punto de montaje**. Por lo general, el punto de montaje es la raíz de un espacio de nombre. Durante la resolución de nombres, se busca el punto de montaje y la resolución procede mediante el acceso a la tabla de directorio.

El principio de montaje se puede generalizar para otros espacios de nombre. En especial, lo que necesitamos es un nodo directorio que actúe como un punto de montaje y almacene toda la información necesaria para identificar y acceder al punto de montaje del espacio de nombres externo. Este método se sigue en numerosos sistemas de archivos distribuidos.

Consideremos una colección de espacios de nombre que está distribuida a través de diferentes máquinas. En especial, cada espacio de nombre se implementa mediante un servidor diferente, ejecutándose cada uno posiblemente en una máquina separada. Por consecuencia, si queremos montar un espacio de nombres externo NS_2, dentro de un espacio de nombre NS_1, pudiera ser necesario comunicarse sobre una red con el servidor de NS_2, mientras que dicho servidor pudiera estar en ejecución sobre una máquina diferente a la del servidor para NS_1. Para montar un espacio de nombres externo dentro de un sistema distribuido se requiere al menos la siguiente información:

1. El nombre de un protocolo de acceso.
2. El nombre de un servidor.
3. El nombre del punto de montaje del espacio de nombres externo.

Observe que cada uno de estos nombres requiere ser resuelto. El nombre de un protocolo de acceso requiere resolverse con la implementación de un protocolo mediante el cual la comunicación con el servidor del espacio de nombre externo pueda tener lugar. El nombre del servidor requiere resolverse para una dirección en donde se pueda alcanzar a dicho servidor. Como la última parte en la resolución de nombres, el nombre del punto de montaje necesita resolverse hacia un identificador de nodo en el espacio de nombre externo.

En sistemas no distribuidos, pudiera ser que ninguno de estos tres puntos sea realmente necesario. Por ejemplo, en UNIX, no existe protocolo de control de acceso y no existe servidor. Además, no se necesita el nombre del punto de montaje dado que es simplemente el directorio raíz del espacio de nombre externo.

El nombre del punto de montaje lo resuelve el servidor del espacio de nombre externo. Sin embargo, también necesitamos espacios de nombre e implementaciones para el protocolo de acceso y el nombre del servidor. Una posibilidad es representar los tres nombres listados antes como una URL.

Para establecer puntos concretos, considere una situación en la cual un usuario con una computadora portátil desea tener acceso a archivos almacenados en un servidor de archivos remoto. La máquina cliente y el servidor de archivos están configurados mediante el **Sistema de Archivos de Red** (**NFS**, por sus siglas en inglés), el cual explicaremos con detalle en el capítulo 11. NFS es un sistema de archivos distribuido que viene con un protocolo que describe de manera precisa la manera en que el cliente puede acceder a un archivo almacenado en un servidor de archivos NFS (remoto). En especial, para permitir a NFS trabajar a lo largo de internet, un cliente puede especificar de manera exacta a cuál archivo desea acceder por medio de una URL en NFS, por ejemplo, *nfs://flits.cs.vu.nl//home/steen*. Esta URL nombra a un archivo (el cual sucede que es un directorio) llamado */home/steen* en el servidor de archivos NFS *flits.cs.vu.nl*, al cual puede acceder un cliente por medio del protocolo NFS (Shepler y cols., 2003).

El nombre *nfs* es conocido en el sentido de que existe un acuerdo mundial para interpretarlo. Dado que tratamos con una URL, el nombre *nfs* se resolverá para una implementación del protocolo NFS. El nombre del servidor es resuelto para su dirección mediante el uso de DNS, el cual explicaremos en la siguiente sección. Como dijimos, */home/steen* se resuelve mediante el servidor del espacio de nombre externo.

La organización de un sistema de archivos en la máquina cliente se muestra parcialmente en la figura 5-12. El directorio raíz tiene el número de las entradas definidas por el usuario, incluso un subdirectorio llamado */remote*. Para este subdirectorio, se intenta la inclusión de puntos de montaje para espacios de nombre externos tales como el directorio de inicio del usuario en la Universidad de Vrije. Hasta este punto, se utiliza un nodo de directorio llamado */remote/vu* para almacenar la URL *nfs://flits.cs.vu.nl//home/steen*.

Consideremos ahora el nombre */remote/vu/mbox*. Este nombre se resuelve al comenzar en el directorio raíz en la máquina del cliente y continuar hasta que se alcanza el nodo */remote/vu*. El proceso de resolución de nombres continúa entonces con la devolución de la URL *nfs://flits.cs.vu.nl //home/steen*, que a su vez propicia que la máquina cliente contacte al servidor de archivos *flits.cs.vu.nl* por medio del protocolo NFS, y subsecuentemente accede al directorio */home/steen*. La resolución de nombres puede continuar mediante la lectura del archivo llamado *mbox* en ese directorio, después de lo cual el proceso de resolución se detiene.

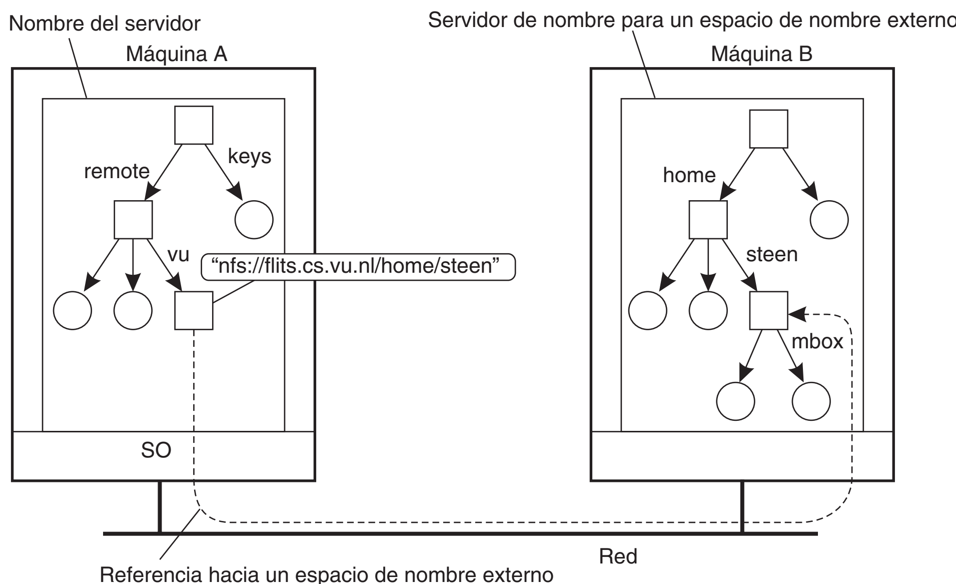


Figura 5-12. Montaje de espacios de nombre remotos a través de protocolos específicos de acceso.

Los sistemas distribuidos que permiten el montaje de un archivo remoto como el que acabamos de describir permiten a la máquina del cliente, por ejemplo, ejecutar los siguientes comandos:

```
cd /remote/vu
ls −l
```

los cuales posteriormente listan los archivos en el directorio */home/steen* en el servidor de archivos remoto. La belleza de todo esto es que el usuario ahorra los detalles del acceso real hacia el servidor remoto. De manera ideal, solamente se nota alguna pérdida de rendimiento comparado con el acceso a los archivos disponibles de manera local. En efecto, para el cliente pareciera que el espacio de nombre ruteado en la máquina local, y el ruteado en */home/steen* en la máquina remota, forman un solo espacio de nombre.

5.3.3 Implementación de un espacio de nombre

Un espacio de nombre da forma al corazón de un servicio de nombres, esto es, un servicio que permite a los usuarios y procesos agregar, quitar, y buscar nombres. Un servicio de nombres se implementa mediante servidores de nombre. Si un sistema distribuido está restringido a una red de área local, con frecuencia es factible implementar un servicio de nombres //revisar (s) por medio de un solo servidor de nombre. Sin embargo, en los sistemas distribuidos implementados a gran escala con muchas entidades, y posiblemente dispersos a lo largo de una gran área geográfica, es necesario distribuir la implementación del espacio de nombre sobre múltiples servidores de nombre.

Distribución de los espacios de nombre

Los espacios de nombre para un sistema distribuido de gran escala, posiblemente a nivel mundial, son organizados por lo general de manera jerárquica. Igual que antes, asuma dicho espacio de nombre como un solo nodo raíz. Para implementar de manera efectiva dicho espacio de nombre, es conveniente colocarlo dentro de capas lógicas. Cheriton y Mann (1989) distinguen entre las tres capas siguientes:

La **capa global** está formada por los nodos de más alto nivel, esto es, el nodo raíz y otros nodos directorio lógicamente cercanos a la raíz, a saber, sus hijos. Los nodos ubicados en la capa global con frecuencia se caracterizan por su estabilidad, en el sentido de que las tablas de directorio rara vez se modifican. Dichos nodos pudieran representar organizaciones, o grupos de organizaciones, para las cuales se almacenan los nombres dentro del espacio de nombre.

La **capa de administración** está formada por los nodos directorio que son administrados juntos dentro de una sola organización. Una característica de los nodos directorio ubicados en la capa de administración es que representan grupos de entidades que pertenecen a la misma organización o a una unidad de administración. Por ejemplo, en una organización pudiera existir un nodo directorio para cada departamento, o un nodo directorio a partir del cual se pueden encontrar todos los servidores. Otro nodo directorio pudiera ser utilizado como punto de inicio para nombrar a todos los usuarios, y así sucesivamente. Los nodos de la capa de administración son relativamente estables, aunque por lo general los cambios ocurren con más frecuencia que en los nodos de la capa global.

Por último, la **capa de dirección** consta generalmente de nodos que pudieran modificarse de manera regular. Por ejemplo, en la red local los nodos representan servidores que pertenecen a esta capa. Por la misma razón, la capa incluye nodos que representan archivos compartidos tales como aquellos implementados para bibliotecas o binarios. Otra clase importante de nodos incluye los que representan directorios definidos por el usuario y archivos. Al contrario de las capas global y de administración, los nodos de la capa de dirección se administran no solamente por administradores de sistemas, sino también por usuarios individuales de un sistema distribuido.

Para hacer los puntos más concretos, la figura 5-13 muestra un ejemplo del particionamiento del espacio de nombre DNS, incluyendo los nombres de archivos localizados dentro de una organización y a los que se puede acceder a través de internet; por ejemplo, páginas web y archivos transferibles. El espacio de nombre está dividido en partes que no se traslapan, llamadas **zonas** en DNS (Mockapetris, 1987). Una zona es parte del espacio de nombre implementado mediante un servidor de nombres separado. Algunas de estas zonas se ilustran en la figura 5-13.

Si damos un vistazo a la disponibilidad y al rendimiento, vemos que en cada capa los servidores de nombre tienen que cumplir con diferentes requerimientos. En la capa global, la alta disponibilidad es especialmente crítica para los servidores de nombre. Si un servidor de nombre falla, una gran parte del espacio de nombre será inalcanzable debido a que la resolución de nombre no puede proseguir más allá del servidor defectuoso.

De alguna manera el rendimiento es sutil. Debido a la baja tasa de cambio de nodos en la capa global, los resultados de las operaciones de búsqueda generalmente permanecen válidos por largo tiempo. En consecuencia, dichos resultados pueden depositarse en caché de manera efectiva (es decir, almacenados localmente) por los clientes. La siguiente vez que se realice la misma operación

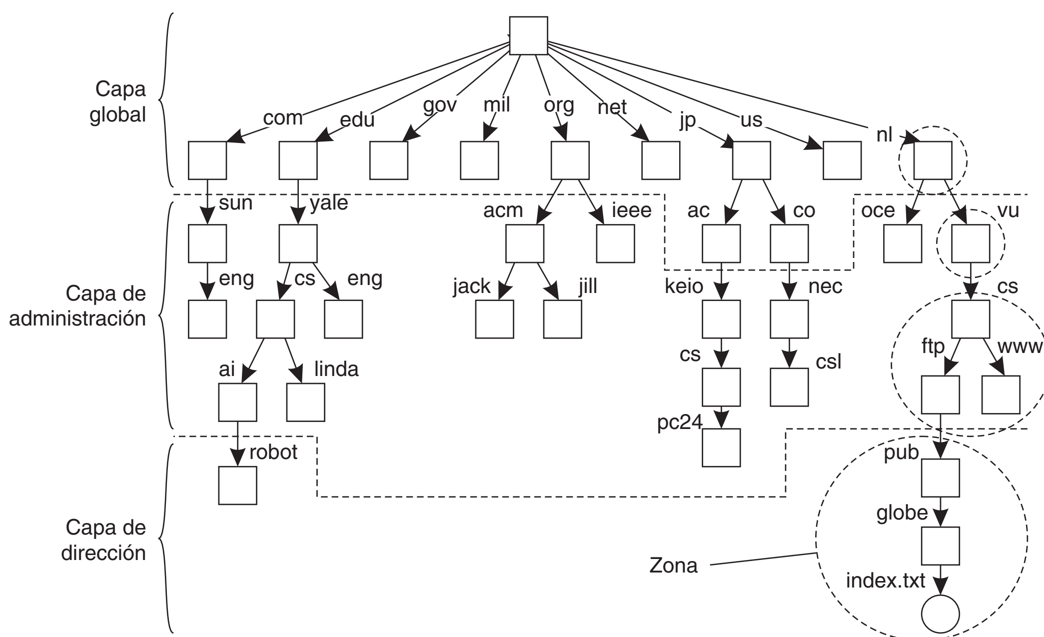


Figura 5-13. Ejemplo de la partición del espacio de nombre DNS, que incluye archivos accesibles mediante internet, dentro de tres capas.

de búsqueda, los datos se pueden recuperar desde el caché del cliente en lugar de dejar que el servidor de nombre devuelva los resultados. Como resultado, en la capa global los servidores de nombre no tienen que responder rápidamente a una simple petición de búsqueda. Por otro lado, el rendimiento de proceso pudiera ser importante, especialmente en sistemas a gran escala con millones de usuarios.

En la capa global, los requerimientos de disponibilidad y rendimiento para los servidores de nombre pueden cumplirse mediante la réplica de servidores, en combinación con el uso del caché del lado del cliente. Tal como explicaremos en el capítulo 7, por lo general no es necesario que en esta capa las actualizaciones surtan un efecto inmediato, lo cual vuelve mucho más fácil mantener réplicas consistentes.

La disponibilidad de un servidor de nombre en la capa de administración es primordialmente importante para los clientes ubicados en la misma organización que la del servidor de nombre. Si éste falla, muchos recursos localizados dentro de la organización se vuelven inalcanzables debido a que no se pueden buscar. Por otro lado, que en una organización los recursos sean temporalmente inalcanzables pudiera ser menos importante para los usuarios externos de dicha organización.

Con respecto al rendimiento, los servidores de nombre implementados en la capa de administración tienen características similares a las de la capa global. Debido a que los cambios a los nodos no ocurren con tanta frecuencia, el resultado del uso del caché de búsquedas puede ser altamente efectivo, lo que vuelve al rendimiento menos crítico. Sin embargo, al contrario de la capa global, la capa de administración debe cuidar que los resultados de las búsquedas se devuelvan en pocos

milisegundos, ya sea de manera directa desde el servidor o desde el caché local del cliente. En forma similar, por lo general las actualizaciones se pueden procesar más rápido que en la capa global. Por ejemplo, es inaceptable que tome horas habilitar una cuenta para un nuevo usuario.

Con frecuencia estos requerimientos se pueden cumplir mediante el uso de máquinas de alto rendimiento para ejecutar servidores de nombre. Además, se debe aplicar el uso del caché del lado del cliente, combinado con replicación para incrementar la disponibilidad general.

Los requerimientos de disponibilidad para los servidores de nombre, en el nivel de administración, por lo general son menos demandantes. En especial, con frecuencia es suficiente el uso de una sola máquina (dedicada) para ejecutar los servidores de nombre en riesgo de no disponibilidad temporal. Sin embargo, el rendimiento es crucial. Los usuarios pueden esperar que las operaciones tengan lugar de inmediato. Debido a que las actualizaciones ocurren de manera regular, el uso del caché del lado del cliente a menudo resulta menos efectivo a menos que se tomen medidas especiales, las cuales explicaremos en el capítulo 7.

Elemento	Global	Administración	Dirección
Escala geográfica de una red	A nivel mundial	Organización	Departamento
Número total de nodos	Pocos	Muchos	Cuantiosos números
Respuesta a las búsquedas	Segundos	Milisegundos	Inmediata
Propagación de actualizaciones	Lenta	Inmediata	Inmediata
Número de réplicas	Muchas	Ninguna o pocas	Ninguna
¿Se aplica el cacheo del lado del cliente?	Sí	Sí	Ocasionalmente

Figura 5-14. Comparación entre servidores de nombre para la implementación de nodos desde un espacio de nombre de gran escala particionado en una capa global, una de administración, y una de dirección.

En la figura 5-14 se muestra una comparación entre servidores de nombre en diferentes capas. En los sistemas distribuidos, los servidores de nombre ubicados en las capas global y de administración son los más difíciles de implementar. Las dificultades son ocasionadas por la replicación y el uso del caché, necesarios para la disponibilidad y el rendimiento, pero que además generan problemas de consistencia. Algunos problemas se agravan debido a que los cachés y las réplicas se distribuyen a lo largo de redes de área amplia, las cuales generan grandes retardos de comunicación que, de esa manera, vuelven mucho más difícil la sincronización. Explicaremos la replicación y el uso del caché extensamente en el capítulo 7.

Implementación de la resolución de nombre

La distribución de un espacio de nombre a lo largo de múltiples servidores de nombre afecta la implementación de la resolución de nombre. Para explicar la implementación de la resolución de nombre en servicios de nombre a gran escala, asumimos por el momento que los servidores de

nombre no están replicados y que no se utilizan cachés del lado del cliente. Cada cliente tiene acceso a un **solucionador de nombre**, el cual es responsable de asegurar que el proceso de resolución de nombre se lleve a cabo. Tomando como referencia la figura 5-13, asumimos que se debe resolver el siguiente nombre de ruta (absoluto).

root:<n1, vu, cs, ftp, pub, globe, index.html>

Mediante el uso de la notación URL, este nombre de ruta correspondería a *ftp://ftp.cs.vu.nl/pub/globe/index.html*. Ahora existen dos maneras de implementar la resolución de nombre.

En una **resolución iterativa de nombre**, un solucionador de nombre toma el nombre completo hacia la raíz del servidor de nombre. Asumimos que la dirección en donde podemos contactar al servidor de nombre es conocida. El servidor raíz solucionará el nombre de ruta hasta donde le sea posible, y devolverá el resultado al cliente. En nuestro ejemplo, el servidor raíz puede resolver solamente la etiqueta *nl*, para lo cual devolverá la dirección de su servidor de nombre asociado.

En este punto, el cliente pasa el resto del nombre de ruta (es decir, *nl:<vu, cs, ftp, pub, globe, index.html>*) a dicho servidor de nombre. Este servidor puede resolver solamente la etiqueta *vu*, y devuelve la dirección del servidor de nombre asociado, junto con el resto del nombre de ruta *vu:<cs, ftp, pub, globe, index.html>*.

Entonces el solucionador del nombre de cliente hará contacto con el siguiente servidor de nombre, el cual responde mediante la solución de la etiqueta *cs*, y en consecuencia también *ftp*, y devuelve la dirección del servidor FTP junto con el nombre de ruta *ftp:<pub, globe, index.html>*. Entonces el cliente contacta al servidor FTP, solicitando la solución de la última parte del nombre de ruta original. El servidor FTP solucionará entonces las etiquetas *pub*, *globe* e *index.html*, y transfiere el archivo solicitado (usando en este caso al servidor FTP). Este proceso de la solución iterativa de nombre aparece en la figura 5-15. (La notación #<*cs*> es utilizada para indicar la dirección del servidor responsable del manejo del nodo al que se hace referencia mediante <*cs*>.)

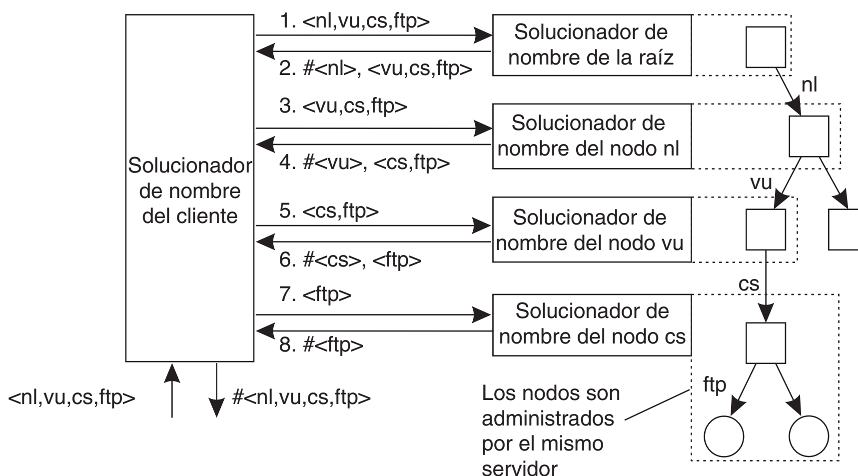


Figura 5-15. El principio de la resolución iterativa de nombre.

En la práctica el último paso, a saber, el contacto del servidor FTP y la solicitud de que transfiera el archivo con el nombre de ruta *ftp:<pub, globe, index.html>*, se realiza de manera separada mediante el proceso del cliente. En otras palabras, normalmente el cliente sólo manejaría el nombre de ruta *root:<nl, vu, cs, ftp>* para el solucionador de nombre, a partir de lo cual pudiéramos esperar la dirección desde donde podemos contactar al servidor FTP, tal como aparece en la figura 5-15.

Una alternativa a la resolución iterativa de nombre es el uso de la recursividad durante la resolución del nombre. En lugar de devolver cada resultado inmediato de nuevo al solucionador de nombre del cliente, mediante la **resolución recursiva de nombre**, un servidor de nombre pasa el resultado al siguiente servidor de nombre. Así, por ejemplo, cuando el servidor de nombre raíz encuentra la dirección del servidor de nombre que implementa el nodo llamado *nl*, solicita al servidor de nombre que resuelva el nombre de la ruta *nl:<vu, cs, ftp, pub, globe, index.html>*. También mediante el uso de la resolución recursiva de nombre, este siguiente servidor resolverá la ruta completa y en algún momento devolverá el archivo *index.html* al servidor raíz, a su vez, pasará ese archivo al solucionador de nombre del cliente.

En la figura 5-16 se muestra la resolución recursiva de nombre. Tal como la resolución iterativa de nombre (al contactar el servidor FTP y pedirle la transferencia del archivo indicado), por lo general se lleva a cabo como un proceso separado del cliente.

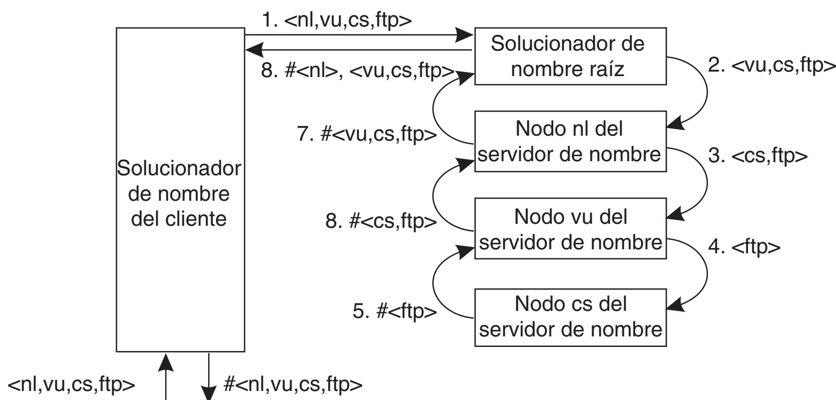


Figura 5-16. El principio de la resolución recursiva de nombre.

La principal desventaja de la resolución recursiva de nombre es que demanda un rendimiento más alto de cada servidor de nombre. De manera básica, se requiere un servidor de nombre para manipular la resolución completa del nombre de ruta, aunque lo haga en cooperación con otros servidores de nombre. Esta dificultad adicional por lo general es tan importante que los servidores de nombre ubicados en la capa global de un espacio de nombre soportan solamente la resolución iterativa de nombre.

Existen otras ventajas importantes en la resolución recursiva de nombres. La primera ventaja es que el uso del caché resulta más efectivo comparado con el de la resolución iterativa de nombre. La segunda es que los costos de la comunicación son más reducidos. Para explicar estas ventajas,

asuma que el solucionador de nombre del cliente aceptará los nombres de ruta que solamente hacen referencia a los nodos localizados en las capas global o de administración del espacio de nombre. Para resolver esa parte del nombre de ruta que corresponde a los nodos de la capa de dirección, un cliente hará contacto de manera separada con el servidor de nombre devuelto por su solucionador de nombre, tal como ya lo explicamos.

La resolución recursiva de nombre permite a cada servidor de nombre aprender gradualmente la dirección de cada servidor de nombre responsable de la implementación de los nodos de más bajo nivel. Como resultado, puede utilizarse el caché para aumentar el rendimiento de manera efectiva. Por ejemplo, cuando a un servidor raíz se le solicita resolver el nombre de ruta *root:<nl, vu, cs, ftp>*, en algún momento obtendrá la dirección del servidor de nombre que implementa el nodo referido por dicho nombre de ruta. Para llegar a este punto, el servidor de nombre para el nodo *nl* tiene que revisar la dirección del servidor de nombre para el nodo *vu*, mientras que este último servidor tiene que revisar la dirección del servidor de nombre que manipula el nodo *cs*.

Debido a que los cambios a los nodos de las capas global y de administración no ocurren a menudo, el servidor de nombre raíz puede almacenar en caché la dirección devuelta de manera efectiva. Más aún, ya que también se devuelve la dirección, mediante recursividad, hacia el servidor de nombre responsable de la implementación del nodo *vu* y al que implementa el nodo *nl*, también pudiera usarse caché a dichos servidores.

De manera similar, también se pueden devolver y guardar en caché los resultados de búsquedas de nombres intermedios. Por ejemplo, el servidor para el nodo *nl* tendrá que buscar la dirección del servidor del nodo *vu*. Dicha dirección se puede devolver al servidor raíz cuando el servidor *nl* devuelve el resultado de la búsqueda del nombre original. En la figura 5-17 podemos ver un resumen completo del proceso de resolución y los resultados que se pueden almacenar en caché para cada servidor de nombre.

Servidor para el nodo	**Debiera resolver**	**Busca**	**Lo pasa al hijo**	**Lo recibe y cachea**	**Lo devuelve al solicitante**
cs	<ftp>	#<ftp>	–	–	#<ftp>
vu	<cs,ftp>	#<cs>	<ftp>	#<ftp>	#<cs> #<cs, ftp>
nl	<vu,cs,ftp>	#<vu>	<cs,ftp>	#<cs> #<cs,ftp>	#<vu> #<vu,cs> #<vu,cs,ftp>
Raíz	<nl,vu,cs,ftp>	#<nl>	<vu,cs,ftp>	#<vu> #<vu,cs> #<vu,cs,ftp>	#<nl> #<nl,vu> #<nl,vu,cs> #<nl,vu,cs,ftp>

Figura 5-17. Resolución recursiva de nombre para *<nl, vu, cs, ftp>*. Resultados intermedios del caché del servidor de nombre para búsquedas subsecuentes.

El principal beneficio de este método es que, de alguna manera, las operaciones de búsqueda se pueden manipular eficientemente. Por ejemplo, supongamos que otro cliente solicita más tarde

la resolución del nombre de ruta *root:<nl, vu, cs, flits>*. Este nombre se pasa a la raíz, la cual de inmediato puede reenviarla al servidor de nombre para el nodo *cs*, y pedir que se resuelva el resto del nombre de ruta *cs:<flits>*.

Con la resolución iterativa de nombre, el uso del caché necesariamente está restringido al solucionador de nombre del cliente. En consecuencia, si el cliente *A* solicita la resolución de un nombre, y luego otro cliente *B* solicita la resolución del mismo nombre, la resolución de nombres tendrá que pasar a través del mismo servidor de nombre tal como lo hizo el cliente *A*. Como un compromiso, muchas organizaciones utilizan un servidor de nombre local e intermedio que se comparte con todos los clientes. Este servidor de nombre local manipula todas las solicitudes de nombre y realiza el almacenamiento en caché de los resultados. Tal servidor intermedio también es conveniente desde el punto de vista directivo. Por ejemplo, sólo ese servidor requiere saber la manera en que se localiza al servidor de nombre raíz; otras máquinas no requieren esta información.

La segunda ventaja de la resolución recursiva de nombre es que a menudo resulta más barata con respecto a la comunicación. De nuevo, consideremos la resolución del nombre de ruta *root:<nl, vu, cs, ftp>* y asumamos que el cliente se localiza en San Francisco. Suponiendo que el cliente conoce la dirección del servidor implementado para el nodo *nl*, mediante la resolución recursiva de nombre, la comunicación sigue la ruta desde el servidor del cliente en San Francisco hasta el servidor *nl* ubicado en Holanda, que se muestra como *R*1 en la figura 5-18. Desde este punto, posteriormente se necesita la comunicación entre el servidor *nl* y el servidor de nombre de la Vrije Universiteit ubicado en el campus universitario de Amsterdam, en Holanda. En la figura 5-18 esta comunicación se indica como *R*2. Finalmente, es necesaria la comunicación entre el servidor *vu* y el servidor de nombre del departamento de ciencias de la computación, que aparece como *R*3. La ruta para enviar la respuesta es la misma, pero en dirección opuesta. De manera clara, los costos de comunicación los dicta el intercambio de mensajes dado entre los servidores del cliente y el servidor *nl*.

Por contraste, con la resolución iterativa de nombre, el servidor del cliente tiene que comunicarse de manera separada con el servidor *nl*, el servidor *vu*, y el servidor *cs*, por lo cual los costos totales pudieran superar cuando mucho en tres veces la resolución recursiva de nombre. En la figura 5-18, las flechas etiquetadas como *I*1, *I*2, e *I*3 muestran la ruta de comunicación para la resolución iterativa de nombre.

5.3.4 Ejemplo: El sistema de nombres de dominio

Uno de los servicios distribuidos de nombres más grande actualmente en uso es el servicio de nombres de dominio en internet (DNS, por sus siglas en inglés). El DNS es primordialmente utilizado para la búsqueda de direcciones IP de servidores y servidores de correo. En las páginas siguientes nos concentraremos en la organización del espacio de nombres del DNS, y en la información almacenada en sus nodos. Además, daremos un vistazo más de cerca a la implementación real de un DNS. Podemos encontrar más información en Mockapetris (1987) y Albitz y Liu (2001). Una valoración reciente del DNS, ligada notablemente a la condición de ajustarse a las necesidades de la internet actual, la podemos encontrar en Levien (2005). A partir de este estudio, es posible esbozar un tanto sorprendente la conclusión de que, incluso después de más de 30 años, el DNS no

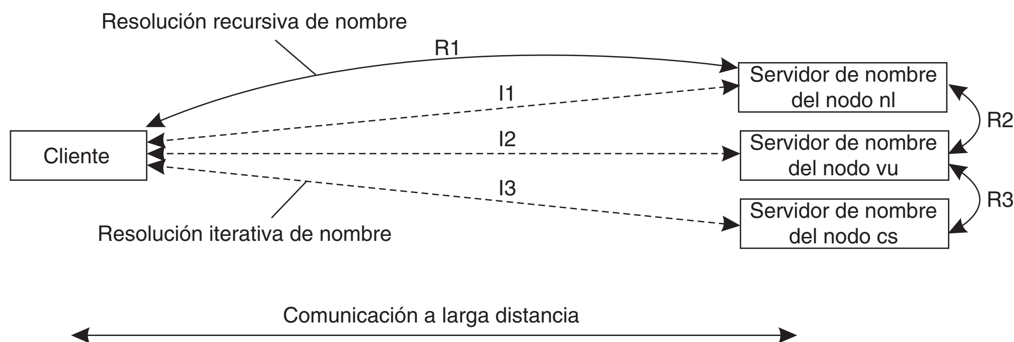


Figura 5-18. Comparación entre las resoluciones recursiva e iterativa de nombre con respecto a los costos de comunicación.

da indicación alguna de que sea necesario reemplazarlo. Podríamos argumentar que la causa principal radica en la profunda comprensión del diseñador acerca de la manera de mantener las cosas simples. En otros campos de los sistemas distribuidos, la práctica indica que no todos los diseñadores están dotados de tal entendimiento.

El espacio de nombre DNS

El espacio de nombre DNS está organizado jerárquicamente como las raíces de un árbol. Una etiqueta es una cadena indiferente al uso de mayúsculas o minúsculas formada por caracteres alfanuméricos. Una etiqueta tiene longitud máxima de 63 caracteres; la longitud del nombre de ruta completo está restringida a 255 caracteres. La representación de la cadena de un nombre de ruta consta de una lista de sus etiquetas, que comienza por la extrema derecha, y separa las etiquetas mediante un punto ("."). La raíz se representa mediante un punto. Así, por ejemplo, el nombre de ruta *root:,nl, vu, cs, flits.*, está representado mediante la cadena *flits.cs.vu.nl*, la cual incluye el punto a la extrema derecha para indicar el nodo raíz. Por lo general se omite este punto para mejor legibilidad.

Debido a que cada nodo localizado en el espacio de nombre DNS tiene exactamente una arista entrante (con excepción del nodo raíz, el cual no contiene aristas entrantes), la etiqueta adjunta a la arista entrante del nodo también se utiliza como el nombre de dicho nodo. A un subárbol se le llama **dominio**; a un nombre de ruta hacia su nodo raíz se le llama **nombre de dominio**. Observe que, tal como un nombre de ruta, un nombre de dominio puede ser absoluto o relativo.

El contenido de un nodo está formado por una colección de **registros de recurso**. Existen diferentes tipos de registros de recurso. Los más importantes aparecen en la figura 5-19.

En el espacio de nombre DNS, un nodo representa a menudo diversas entidades al mismo tiempo. Por ejemplo, un nombre de dominio tal como *vu.nl* se utiliza para representar un dominio y una zona. En este caso, el dominio se implementa por medio de distintas zonas (no sobrepuestas).

Un registro de recurso *SOA* (*start of authority*; principio de autoridad) contiene información tal como una dirección de correo electrónico del administrador de sistema responsable de la zona representada, el nombre del servidor desde donde se recuperan los datos con respecto a la zona, y así sucesivamente.

Tipo de registro	Entidad asociada	Descripción
SOA	Zona	Mantiene la información con respecto a la zona representada
A	Servidor	Contiene una dirección IP acerca del servidor que representa a este nodo
MX	Dominio	Hace referencia a un servidor de correo electrónico para manipular la dirección de correo de este nodo
SRV	Dominio	Hace referencia a un servidor que manipula un servicio específico
NS	Zona	Hace referencia a un servidor de nombre que implementa al nodo representado
CNAME	Nodo	Vínculo simbólico con el nodo primario del nodo representado
PTR	Servidor	Contiene el nombre canónico del servidor
HINFO	Servidor	Mantiene información referente al servidor que representa a este nodo
TXT	Cualquier tipo	Contiene cualquier información específica que se considere de utilidad

Figura 5-19. Tipos de registros de recurso más importantes que forma el contenido de los nodos en el espacio de nombre del DNS.

Un registro (dirección) *A* representa un servidor particular en internet. El registro *A* contiene una dirección IP hacia dicho servidor para permitir la comunicación. Si un servidor contiene diversas direcciones IP, como en el caso de máquinas con diversas direcciones de inicio, el nodo contendrá un registro *A* para cada dirección.

Otro tipo de registro es el registro *MX* (*mail exchange*; intercambio de correo), el cual es como un vínculo simbólico que representa a un servidor de correo. Por ejemplo, el nodo que representa al dominio *cs.vu.nl* contiene un registro *MX* que contiene el nombre *zephyr.cs.vu.nl*, el cual hace referencia a un servidor de correo. Dicho servidor manipulará todos los mensajes entrantes direccionados hacia los usuarios en el dominio *cs.vu.nl*. Pueden existir diversos registros *MX* almacenados dentro de un nodo.

Relacionados con los registros *MX* están los registros *SRV*, los cuales contienen el nombre de un servidor para un servicio específico. Los registros *SRV* están definidos en Gulbrandsen (2000). Por sí mismo, el servicio está identificado mediante un nombre junto con el nombre de un protocolo. Por ejemplo, el servidor web del dominio *cs.vu.nl* pudiera ser nombrado por medio de un registro *SRV* tal como *_http_tcp.cs.vu.nl*. Entonces, este registro haría referencia al nombre real del servidor (el cual es *soling.cs.vu.nl*). Una ventaja importante de los registros SRV es que los clientes ya no requieren conocer el nombre DNS del servidor que proporciona un servicio específico. En vez de eso, únicamente requieren estandarizar los nombres de servicio, después de lo cual se puede bloquear el servidor proveedor.

Los nodos representan una zona, contienen uno o más registros *NS* (servidores de nombre). Tal como los registros *MX*, un registro *NS* contiene el nombre de un servidor de nombre que implementa la zona representada por el nodo. En principio, cada nodo del espacio de nombre se puede almacenar en el registro *NS* que hace referencia al servidor de nombre que lo implementa. Sin embargo, como lo explicaremos más adelante, la implementación de un espacio de nombre DNS es tal que solamente los nodos que representan zonas requieren almacenar registros *NS*.

Un DNS distingue un alias a partir de lo que conocemos como **nombres canónicos**. Se asume que cada servidor tiene un nombre primario o canónico. Un alias se implementa por medio de un

nodo que almacena un registro *CNAME* que contiene el nombre canónico de un servidor. De esta manera, el nombre de un nodo que almacena tal registro es el nombre de un vínculo simbólico, como ilustramos en la figura 5-11.

El DNS mantiene un mapeo inverso de las direcciones IP hacia los nombres de los servidores por medio de registros *PTR* (apuntadores). Para acomodar las búsquedas de los nombres de los servidores cuando solamente se tiene la dirección IP, el DNS mantiene un dominio llamado *in-addr.arpa*, el cual contiene los nodos que representan a los servidores de internet y los cuales se llaman mediante la dirección IP del nodo al que representan. Por ejemplo, el servidor *www.cs.vu.nl* tiene una dirección 130.37.20.20. El DNS crea un nodo llamado *20.20.37.130.in-addr.arpa*, que se utiliza para almacenar el nombre canónico de dicho servidor (el cual sucede que es *soling.cs.vu.nl*) en un registro *PTR*.

Los dos últimos tipos de registro son *HINFO* y *TXT*. Un registro *HINFO* (información del servidor) se utiliza para almacenar información adicional acerca del servidor, tal como su tipo de máquina y su sistema operativo. De manera similar, los registros *TXT* se utilizan para cualquier otro tipo de dato que el usuario encuentre útil para almacenar acerca de la entidad representada por el nodo.

Implementación de un DNS

En esencia, el espacio de nombre DNS se puede dividir dentro de una capa global y una capa de administración, tal como aparece en la figura 5-13. La capa de dirección, que por lo general está conformada por los sistemas de archivos locales, no es parte formal del DNS y, por tanto, tampoco es manejada por éste.

Cada zona se implementa mediante un servidor de nombre, el cual virtualmente se replica siempre para asegurar la estabilidad. Por lo general, las actualizaciones para una zona son manipuladas por el servidor de nombre primario. Las actualizaciones toman lugar mediante la modificación de la base de datos del DNS local al servidor primario. Los servidores de nombre secundarios no tienen acceso directo a la base de datos, pero, en su lugar, solicitan al servidor primario que transfiera su contenido. A esto último se le llama **transferencia de zona** en la terminología DNS.

Una base de datos DNS se implementa como una (pequeña) colección de archivos, de los cuales los más importantes contienen todos los recursos para *todos* los nodos de una zona en particular. Este método permite a los nodos ser identificados empleando simplemente su nombre de dominio, por lo que la idea de un nodo identificador se reduce a un índice (implícito) dentro de un archivo.

Para comprender mejor estos problemas de implementación, la figura 5-20 muestra una pequeña porción del archivo que contiene la mayor parte de la información para el dominio *cs.vu.nl* (para mayor sencillez, el archivo se editó). El archivo muestra el contenido de diversos nodos que son parte del dominio *cs.vu.nl*, donde cada nodo se identifica por medio de su nombre de dominio.

El nodo *cs.vu.nl* representa al dominio así como a la zona. Su registro de recurso *SOA* contiene información específica con respecto a la validez de este archivo, lo cual no abordaremos en este libro. Existen cuatro nombres de servidores para esta zona, a los que se hace referencia mediante su nombre canónico dentro de los registros *NS*. El registro *TXT* se usa para brindar

información adicional con respecto a esta zona, pero no se puede procesar de manera automática por ningún servidor de nombre. Más aún, existe un solo servidor de correo que puede manipular el correo entrante direccionado por usuarios en este dominio. El número que precede al nombre de un servidor de correo especifica la prioridad de selección. Un servidor de correo que envía siempre debe intentar primero hacer contacto con el servidor de correo con el número más pequeño.

Nombre	Tipo de registro	Valor de registro
cs.vu.nl.	SOA	star.cs.vu.nl.hostmaster.cs.vu.nl. 2005092900 7200 3600 2419200 3600
cs.vu.nl.	TXT	"Vrije Universiteit – Math. & Comp. Sc."
cs.vu.nl.	MX	1 mail.few.vu.nl.
cs.vu.nl.	NS	ns.vu.nl.
cs.vu.nl.	NS	top.cs.vu.nl.
cs.vu.nl.	NS	olo.cs.vu.nl.
cs.vu.nl.	NS	star.cs.vu.nl.
star.cs.vu.nl.	A	130.37.24.6
star.cs.vu.nl.	A	192.31.231.42
star.cs.vu.nl.	MX	1 star.cs.vu.nl.
star.cs.vu.nl.	MX	666 zephyr.cs.vu.nl.
star.cs.vu.nl.	HINFO	"Sun" "Unix"
zephyr.cs.vu.nl.	A	130.37.20.10
zephyr.cs.vu.nl.	MX	1 zephyr.cs.vu.nl.
zephyr.cs.vu.nl.	MX	2 tornado.cs.vu.nl.
zephyr.cs.vu.nl.	HINFO	"Sun" "Unix"
ftp.cs.vu.nl.	CNAME	soling.cs.vu.nl.
www.cs.vu.nl.	CNAME	soling.cs.vu.nl.
soling.cs.vu.nl.	A	130.37.20.20
soling.cs.vu.nl.	MX	1 soling.cs.vu.nl.
soling.cs.vu.nl.	MX	666 zephyr.cs.vu.nl
soling.cs.vu.nl.	HINFO	"Sun" "Unix"
vucs-das1.cs.vu.nl.	PTR	0.198.37.130.in-addr.arpa
vucs-das1.cs.vu.nl.	A	130.37.198.0
inkt.cs.vu.nl.	HINFO	"OCE" "Proprietary"
inkt.cs.vu.nl.	A	192.168.4.3
pen.cs.vu.nl.	HINFO	"OCE" "Proprietary"
pen.cs.vu.nl.	A	192.168.4.2
localhost.cs.vu.nl.	A	127.0.0.1

Figura 5-20. Extracto de la base de datos DNS para la zona *cs.vu.nl.*

El servidor *star.cs.vu.nl* opera como un servidor de nombre para esta zona. Los servidores de nombre son críticos para cualquier servicio de nombre. Lo que podemos ver con respecto a este servidor de nombre es que los recursos adicionales se crean para dar dos interfaces de red

por separado, cada una representada por un registro de recurso *A* por separado. De esta manera, los efectos de un vínculo de red roto se puede mitigar en cierta forma si el servidor se mantiene accesible.

Las siguientes cuatro líneas (para *zephyr.cs.vu.nl*) contienen la información necesaria acerca de uno de los servidores de correo del departamento. Observe que este servidor de correo también está respaldado por otro servidor de correo, cuya ruta es *tornado.cs.vu.nl*.

Las siguientes seis líneas muestran una configuración común en la cual el servidor web del departamento, así como el servidor FTP del departamento se implementan mediante una sola máquina, llamada *soling.cs.vu.nl*. Mediante la ejecución de ambos servidores en la misma máquina (y en esencia por el uso de dicha máquina solamente para servicios de internet y nada más), la administración de sistemas se vuelve más fácil. Por ejemplo, ambos servidores tendrán la misma vista del sistema de archivos, y por eficiencia, parte del sistema de archivos pudiera implementarse en *soling.cs.vu.nl*. Con frecuencia este método se aplica en el caso de servicios WWW y FTP.

Las dos siguientes líneas muestran información en uno de los cluster de servidor más viejos del departamento. En este caso, nos indica que la dirección *130.37.198.0* está asociada con el nombre del servidor *vucs-das1.cs.vu.nl*.

Las siguientes cuatro líneas muestran información relativa a dos impresoras grandes conectadas a la red local. Observe que las direcciones ubicadas en el rango de *192.168.0.0* a *192.168.255.255* son privadas: se puede acceder a ellas solamente desde adentro de la red local y no son accesibles desde un servidor de internet arbitrario.

Nombre	Tipo de registro	Valor de registro
cs.vu.nl.	NS	solo.cs.vu.nl.
cs.vu.nl.	NS	star.cs.vu.nl.
cs.vu.nl.	NS	ns.vu.nl.
cs.vu.nl.	NS	top.cs.vu.nl.
ns.vu.nl.	A	130.37.129.4
top.cs.vu.nl.	A	130.37.20.4
solo.cs.vu.nl.	A	130.37.20.5
star.cs.vu.nl.	A	130.37.24.6
star.cs.vu.nl.	A	192.31.231.42

Figura 5-21. Parte de la descripción del dominio *vu.nl* que contiene el dominio *cs.vu.nl*.

Debido a que el dominio *cs.vu.nl* se implementa como una sola zona, en la figura 5-20 no se incluyen las referencias hacia otras zonas. La manera de hacer referencia a nodos de un subdominio que están implementados en una zona diferente se muestra en la figura 5-21. Lo que se necesita hacer es especificar el servidor de nombre para el subdominio al simplemente dar su nombre de dominio y su dirección IP. Al resolver el nombre de un nodo que radica en el dominio *cs.vu.nl*, la resolución de nombre continúa en cierto punto con la lectura de la base de datos DNS almacenada mediante el servidor de nombre para el dominio *cs.vu.nl*.

Implementaciones descentralizadas de DNS

La implementación de un DNS que describimos hasta aquí es la estándar. Sigue una jerarquía de servidores con 13 servidores raíz muy conocidos y finaliza con millones de servidores en las hojas. Una observación importante es que los nodos de más alto nivel reciben muchas más peticiones que los de más bajo nivel. Sólo mediante el uso del caché de los enlaces nombre-dirección de estos niveles más altos es posible evitar el envío de peticiones hacia ellos y de esta manera intercambiarlos.

Todos estos problemas de escalabilidad se pueden evitar mediante soluciones completamente descentralizadas. En particular, podemos calcular el hash de un nombre DNS, y en consecuencia tomar dicho hash como el valor clave a revisar dentro de una tabla hash distribuida o un servicio de localización jerárquico con un nodo raíz completamente particionado. La desventaja evidente de este método es que perdemos la estructura del nombre original. Esta pérdida pudiera prevenir implementaciones eficientes de, por ejemplo, encontrar al hijo dentro de un dominio específico.

Por otro lado, existen muchas ventajas en el mapeo de DNS hacia una implementación basada en DHT, notablemente su escalabilidad. Como explican Walfish y colaboradores (2004), cuando existe la necesidad de tener muchos nombres, el uso de identificadores como una forma semánticamente libre de acceso a los datos permitirá que diferentes sistemas hagan uso de un solo sistema de nombres. La razón es sencilla: por ahora comprendemos bien cómo podemos soportar de manera eficiente una gran colección de nombres (planos). Lo que necesitamos hacer es mantener el mapeo de la información de un identificador hacia un nombre, donde en este caso un nombre pudiera provenir de un espacio DNS, ser una URL, y así sucesivamente. El uso de identificadores se puede hacer más fácil al permitir a usuarios u organizaciones emplear un espacio de nombre estricto. Esto último es completamente análogo a mantener una configuración privada de variables de ambiente en una computadora.

El mapeo DNS dentro de sistemas punto a punto basados en DHT se explora en CoDoNS (Ramasubramanian y Sirer, 2004a). Los CoDoNS utilizan un sistema basado en DHT en el cual los prefijos de llaves se usan para implementar el ruteo hacia un nodo. Para explicarlo, consideremos el caso en que cada dígito de un identificador se toma desde la configuración $\{0,\ldots, b - 1\}$, donde b es el número base. Por ejemplo, en el sistema de cuerdas, $b = 2$. Si asumimos que $b = 4$, entonces considere un nodo cuyo identificador es 3210. En su sistema, se asume que este nodo mantiene una tabla de ruteo de nodos que tiene los siguientes identificadores:

n_0: un nodo cuyo identificador tiene el prefijo 0
n_1: un nodo cuyo identificador tiene el prefijo 1
n_2: un nodo cuyo identificador tiene el prefijo 2
n_{30}: un nodo cuyo identificador tiene el prefijo 30
n_{31}: un nodo cuyo identificador tiene el prefijo 31
n_{33}: un nodo cuyo identificador tiene el prefijo 33
n_{320}: un nodo cuyo identificador tiene el prefijo 320
n_{322}: un nodo cuyo identificador tiene el prefijo 322
n_{323}: un nodo cuyo identificador tiene el prefijo 323

El nodo 3210 es responsable del manejo de llaves que tienen el prefijo 321. Si recibe una petición de búsqueda para la llave 3123, la reenviará al nodo n_{31}, el cual, en cambio, revisará si requiere reenviarla a un nodo cuyo identificador contenga el prefijo 312. (Debemos advertir que cada nodo mantiene dos listas adicionales que se pueden utilizar para el ruteo si se pierde una entrada dentro de su tabla de ruteo.) Los detalles de este método se pueden encontrar en Pastry (Rowstron y Druschel, 2001) y Tapestry (Zhao y cols., 2004).

Regresemos a CoDoNS, donde un nodo responsable de la llave *k* almacena el registro de recurso DNS asociado con el nombre de dominio hashes hacia *k*. Sin embargo, la parte interesante es que CoDoNS intenta minimizar el número de saltos necesarios en una petición de ruteo mediante la réplica de registros de recursos. El principio de esta estrategia es simple: el nodo 3210 replicará su contenido hacia los nodos que tienen el prefijo 321. Dicha replicación reducirá cada ruta que termina con el nodo 3210 mediante un salto. Por supuesto, esta réplica se aplica de nuevo a todos los nodos que contienen el prefijo 32, y así sucesivamente.

Cuando se replica un registro DNS en todos los nodos que tienen *i* prefijos coincidentes, decimos que se replica a nivel *i*. Observe que para encontrar un registro replicado a nivel *i* (por lo general) se requieren *i* pasos de búsqueda. Sin embargo, existe un intercambio entre el nivel de réplica y el uso de una red y los recursos del nodo. Lo que hace CoDoNS es replicar hasta el punto en que la latencia de la agregación de la búsqueda resultante es menor a la constante dada *C*.

De manera más específica, piense por un momento en la distribución de frecuencia de las consultas. Imagine clasificar las consultas de búsqueda mediante la frecuencia con que se solicita una tecla en especial, y colocar la tecla más solicitada en la primera posición. Se dice que la distribución de las búsquedas es de la **forma Zipf** cuando la frecuencia del elemento clasificado en la enésima posición es proporcional a $1/n^{\alpha}$, con α cercana a 1. George Zipf fue un lingüista de Harvard, y descubrió esta distribución mientras estudiaba la frecuencia del uso de las palabras en un lenguaje natural. Sin embargo, tal como se volvió evidente, es aplicable también a muchas otras cosas, entre las que se encuentran la población de las ciudades, la magnitud de los temblores, las distribuciones de los ingresos más altos, las ganancias de las empresas y, probablemente, consultas DNS (Jung y cols., 2002).

Ahora, si x_i es la fracción de los registros más populares que van a replicarse en el nivel *i*, entonces Ramasubramanian y Sirer (2004b) muestran que x_i puede expresarse mediante la siguiente fórmula (para nuestros propósitos, el simple hecho de que esta fórmula exista ya es importante; en un momento veremos cómo utilizarla):

$$x_i = \left[\frac{d^i(logN - C)}{1 + d + \cdots + d^{\log N - 1}} \right]^{\frac{1}{(1-\alpha)}} \quad con \;\; d = b^{(1-\alpha)/\alpha}$$

donde *N* es la cantidad de nodos de la red y α es el parámetro de la distribución Zipf.

Esta fórmula nos permite tomar decisiones informadas sobre cuáles registros DNS deben replicarse. Para concretar, considere el caso en que $b = 32$ y $\alpha = 0.9$. Entonces, en una red con 10 000 nodos y 1 000 000 de registros DNS, e intentando conseguir un promedio de $C = 1$ salto sólo al realizar una búsqueda, tendremos que $x_0 = 0.0000701674$, lo cual significa que solamente los 70 registros más populares DNS deben replicarse en todas partes. De igual manera, con $x_1 = 0.00330605$,

los siguientes 3306 registros más populares deben replicarse en el nivel 1. Por supuesto, se requiere una $x_i < 1$. En este ejemplo, $x_2 = 0.155769$ y $x_3 > 1$, por lo que únicamente los siguientes 155769 registros más populares se replican, y todos los demás no. Sin embargo, en promedio, un solo salto es suficiente para encontrar un registro DNS solicitado.

5.4 NOMBRES BASADOS EN ATRIBUTOS

Los nombres planos y estructurados generalmente proporcionan una forma única e independiente de la ubicación de referencia a entidades. Más aún, los nombres estructurados han sido parcialmente diseñados para proporcionar una forma amigable de nombrar entidades en forma tal que se pueda acceder a ellas de manera conveniente. En la mayoría de los casos, se supone que el nombre se refiere a una sola entidad. Sin embargo, la independencia de la ubicación y lo amable para las personas no son los únicos criterios para asignar nombres a las entidades. En particular, es muy importante tener la mayor información disponible para realizar una búsqueda efectiva de entidades. Este método requiere del usuario que pueda proporcionar una descripción simple de lo que busca.

Existen muchas formas en las que se pueden proporcionar descripciones, pero una muy popular entre los sistemas distribuidos es describir a una entidad en términos de pares (*atributo*, *valor*), y generalmente se les conoce como **nombres basados en atributos**. En este método, se supone que una entidad tiene asociada una colección de atributos. Cada atributo dice algo sobre dicha entidad. Al especificar cuáles valores debe tener un atributo específico, un usuario básicamente restringe el conjunto de entidades que le interesan. Depende del sistema para asignar nombres el devolver una o más entidades que satisfagan la descripción del usuario. En esta sección veremos con detalle los sistemas de nombres basados en atributos.

5.4.1 Servicios de directorio

Los sistemas de nombres basados en atributos también se conocen como **servicios de directorio**, mientras que los sistemas que soportan nombres estructurados generalmente se conocen como **sistemas de nombres**. Con los servicios de directorio, las entidades tienen asociado un conjunto de atributos que puede utilizarse para búsqueda. En algunos casos, elegir atributos puede resultar relativamente simple. Por ejemplo, en un sistema de correo electrónico, los mensajes pueden adjuntarse con atributos para remitente, destinatario, asunto, etc. Sin embargo, incluso en el caso del correo electrónico, elegir atributos se torna difícil cuando se necesitan otros tipos de descriptores, tal como ilustra la dificultad de desarrollar filtros que sólo permitan el paso de cierto tipo de mensajes (basados en sus descriptores).

Todo esto se reduce a que el diseño de un conjunto adecuado de atributos no es algo trivial. En la mayoría de los casos, el diseño de atributos tiene que hacerse manualmente. Incluso cuando existe un consenso sobre el conjunto de atributos a utilizar, la práctica muestra que configurar consistentemente los valores de un grupo de personas es un problema por sí mismo, como lo habrán experimentado muchos al acceder a bases de datos de música y videos en internet.

Para mitigar algunos de estos problemas, la investigación se ha encaminado a unificar las formas en que pueden describirse los recursos. En el contexto de los sistemas distribuidos, un desarrollo particularmente importante es el **marco de descripción de recursos** (**RDF**, por sus siglas en inglés). Lo básico del modelo RDF es que los recursos se describen como tríos que constan de un sujeto, un predicado, y un objeto. Por ejemplo, (*Persona*, *nombre*, *Alice*) describe un recurso *Persona* cuyo *nombre* es *Alice*. En RDF, cada sujeto, predicado y objeto puede ser un recurso por sí mismo. Esto significa que *Alice* puede implementarse como una referencia a un archivo que puede recuperarse posteriormente. En el caso de un predicado, tal recurso podría contener una descripción textual de dicho predicado. Por supuesto, los recursos asociados con sujetos y objetos podrían ser cualquier cosa. Las referencias en RDF son básicamente URL.

Si las descripciones de recursos se almacenan, se hace posible consultar ese almacén de una forma común para muchos sistemas de nombres basados en atributos. Por ejemplo, una aplicación podría solicitar información asociada con una persona llamada Alice. Tal consulta devolvería una referencia al recurso persona asociado con Alice. La aplicación puede entonces buscar este recurso. Podemos encontrar más información sobre RDF en Manola y Miller (2004).

En este ejemplo, las descripciones de recursos se almacenan en una ubicación central. No hay razón por la que los recursos deban residir en la misma ubicación. Sin embargo, no tener las descripciones en el mismo lugar puede ocasionar un serio problema de rendimiento. A diferencia de los sistemas de nombre estructurados, buscar valores en un sistema de nombre basados en atributos básicamente requiere una búsqueda exhaustiva a través de todos los descriptores. Cuando se considera el rendimiento, tal búsqueda es un problema menor dentro de un solo almacén de datos, pero se necesita aplicar técnicas por separado cuando los datos están distribuidos en diversas, y potencialmente dispersas, computadoras. A continuación veremos diferentes métodos útiles para resolver este problema en sistemas distribuidos.

5.4.2 Implementaciones jerárquicas: LDAP

Un método común para abordar los servicios distribuidos de directorio es combinar nombres estructurados con nombres basados en atributos. Este método ha sido ampliamente adoptado, por ejemplo, en el servicio de directorio activo de Microsoft y en otros sistemas. Muchos de estos sistemas utilizan o se basan en el **LDAP** (*lightweight directory access protocol*; **protocolo ligero de acceso a directorios**). El servicio de directorios LDAP se derivó del servicio de directorios X.500 de OSI. Así como sucede con muchos servicios de OSI, la calidad de sus implementaciones asociadas entorpeció que se ampliara su uso, y se necesitaron simplificaciones para volverlo útil. Puede encontrarse información detallada sobre LDAP en Arkills (2003).

En forma conceptual, un servicio de directorio LDAP consta de cierto número de registros, generalmente conocidos como entradas de directorio. Una entrada de directorio es comparable con un registro de recurso en DNS. Cada registro se conforma con una colección de pares (*atributo*, *valor*), donde cada atributo tiene un tipo asociado. Existe una diferencia entre atributos con un solo valor y atributos con diversos valores. Esto último a menudo representa arreglos y listas. Como ejemplo, una sencilla entrada de directorio que identifica las direcciones de red de algunos servidores de la figura 5-20 aparece en la figura 5-22.

Atributo	Abreviatura	Valor
País	P	NL
Localidad	L	Amsterdam
Organización	O	Vrije Universiteit
UnidadOrganizacional	UO	Comp.Sc.
NombreComún	NC	Main server
Servidores_Correo	2	137.37.20.3, 130.37.24.6, 137.37.20.10
Servidor_FTP	2	130.37.20.20
Servidor_WWW	2	130.37.20.20

Figura 5-22. Un sencillo ejemplo de una entrada de directorio LDAP utilizando convenciones de nombre LDAP.

En nuestro ejemplo, utilizamos una convención de nombre descrita en los estándares LDAP, la cual se aplica a los cinco primeros atributos. Los atributos *Organización* y *UnidadOrganizacional* describen, respectivamente, la organización y el departamento asociado con los datos almacenados en el registro. De igual manera, los atributos *Localidad* y *País* proporcionan información adicional sobre dónde se almacena la entrada. El atributo *NombreComún* con frecuencia se utiliza como un nombre (ambigüo) para identificar una entrada dentro de una parte limitada de un directorio. Por ejemplo, el nombre "servidor principal" puede ser suficiente para encontrar la entrada de nuestro ejemplo, dados los valores específicos de los otros cuatro atributos *País*, *Localidad*, *Organización* y *UnidadOrganizacional*. En nuestro ejemplo, sólo el atributo Servidores_Correo tiene varios valores asociados. Todos los demás atributos sólo tienen un valor.

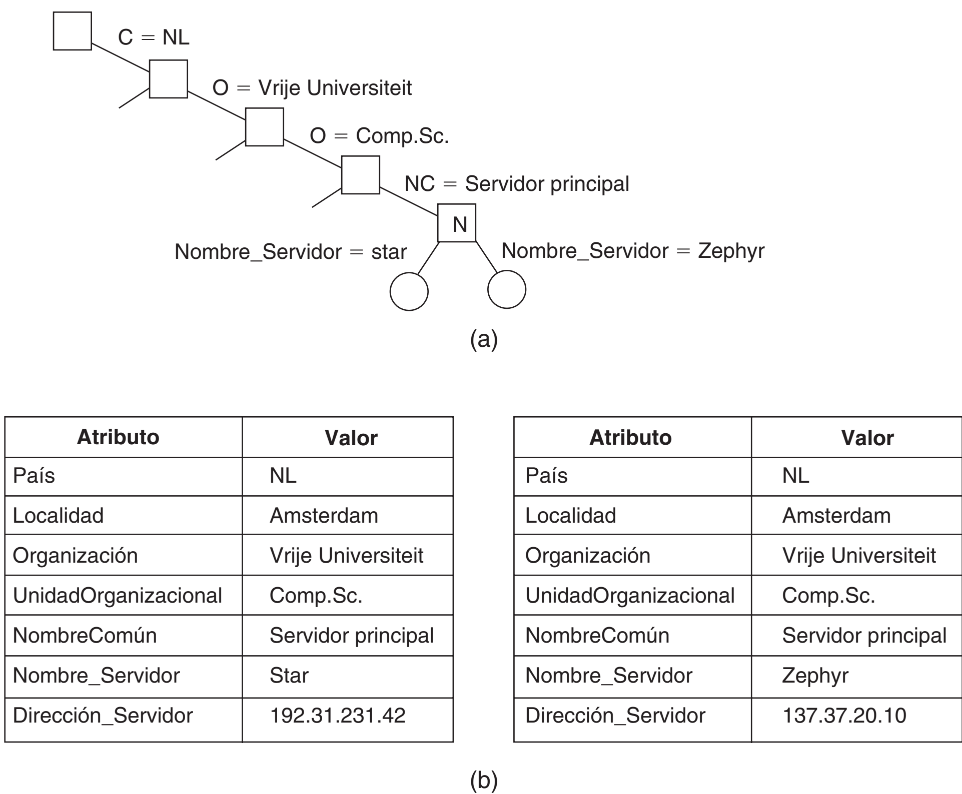
La colección de todas las entradas de directorio en un servicio de directorio LDAP se conoce como **directorio de información base** (**DIB**, por sus siglas en inglés). Un aspecto importante de una DIB es que cada registro tiene un nombre único para que pueda ser buscado. Tal nombre globalmente único aparece como una secuencia de atributos de nombre en cada registro. Cada atributo de nombre se conoce como **nombre relativo diferenciado**, o **RDN** (por sus siglas en inglés). En nuestro ejemplo de la figura 5-22, los primeros cinco atributos son atributos de nombre. Al utilizar las abreviaturas convencionales para representar atributos de nombre en LDAP, como ilustra la figura 5-22, los atributos *País*, *Organización* y *UnidadOrganizacional* pueden utilizarse para formar el nombre globalmente único

/C=NL/O=Vrije Universiteit/UO=Comp.Sc.

análogo al nombre DNS *nl.vu.cs*.

Como en DNS, el uso de nombres globalmente únicos listando RDN en secuencia conduce a una colección jerárquica de entradas de directorio, lo cual se conoce como **árbol de información de directorio (DIT**, por sus siglas en inglés). Un DIT forma, esencialmente, el grafo de nombres de un servicio de directorio LDAP, en la que cada nodo representa una entrada de directorio. Además,

un nodo puede actuar también como un directorio en el sentido tradicional, ya que puede haber varios hijos para los que el nodo actúe como padre. Para explicarlo, consideremos el grafo de nombres tal como se muestra parcialmente en la figura 5-23(a). (Recuerde que las etiquetas están asociadas a las aristas.)



Atributo	Valor
País	NL
Localidad	Amsterdam
Organización	Vrije Universiteit
UnidadOrganizacional	Comp.Sc.
NombreComún	Servidor principal
Nombre_Servidor	Star
Dirección_Servidor	192.31.231.42

Atributo	Valor
País	NL
Localidad	Amsterdam
Organización	Vrije Universiteit
UnidadOrganizacional	Comp.Sc.
NombreComún	Servidor principal
Nombre_Servidor	Zephyr
Dirección_Servidor	137.37.20.10

(b)

Figura 5-23. (a) Parte de un árbol de información de directorio. (b) Dos entradas de directorio que tienen *Nombre_Servidor* como RDN.

El nodo *N* corresponde a la entrada de directorio que aparece en la figura 5-22. Al mismo tiempo, este nodo actúa como padre de otras entradas de directorio que tienen un atributo de nombre adicional, *Nombre_Servidor*, que se utiliza como RDN. Por ejemplo, tales entradas pueden utilizarse para representar servidores, como indica la figura 5-23(b).

Un nodo en un grafo de nombres LDAP puede representar simultáneamente a un directorio en el sentido tradicional que explicamos anteriormente y a un registro LDAP. Esta diferencia es soportada por dos operaciones de búsqueda diferentes. La operación read es utilizada para leer un solo registro dado su nombre de ruta en el DIT. Por contraste, la operación list se utiliza para listar los nombres de todas las aristas de salida de un nodo dado en el DIT. Cada nombre corresponde a un

nodo hijo del nodo dado. Observe que la operación list no devuelve ningún registro; simplemente devuelve nombres. En otras palabras, llamar a read con el nombre de entrada

/C=NL/O=Vrije Universiteit/UO=Comp.Sc./NC=Servidor principal

devolverá el registro que aparece en la figura 5-22, mientras que llamar a list devolverá los nombres *Star* y *Zephyr* a partir de las entradas que muestra la figura 5-23(b), así como los nombres de otros servidores que se han registrado de manera similar.

Para implementar un servicio de directorio LDAP se procede de manera muy similar a la de implementar un servicio de nombres como DNS, con excepción de que LDAP soporta más operaciones de búsqueda, según explicaremos en breve. Cuando se maneja un directorio a gran escala, el DIT generalmente se particiona y distribuye a través de varios servidores conocidos como **agentes de servicios de directorio** (**DSA**, por sus siglas en inglés). Cada parte de un DIT particionado corresponde a una zona del DNS. De igual manera, cada DSA se comporta en forma muy similar a un servidor de nombres normal, excepto que éste implementa cierto número de servicios típicos de directorio, tales como operaciones de búsqueda avanzadas.

Los clientes son representados por lo que se conoce como **servicios de agentes de directorio**, o simplemente **DUA**. Un DUA es parecido a un solucionador de nombre en los servicios de nombres estructurados. Un DUA intercambia información con un DSA de acuerdo con un protocolo de acceso estandarizado.

Lo que diferencia una implementación DLAP de una DNS son las herramientas para buscar a través de una DIB. En particular, las herramientas son proporcionadas para buscar una entrada de directorio dado un conjunto de criterios que los atributos de la entrada buscada deben satisfacer. Por ejemplo, suponga que queremos una lista de los principales servidores de la Vrije Universiteit. Si utilizamos la notación definida en Howes (1997), dicha lista puede ser devuelta utilizando una operación de búsqueda tal como

Respuesta = búsqueda("&(C = NL)(O = Vrije Universiteit)(UO = *)(NC = Servidor principal)")

En este ejemplo, especificamos que el lugar para buscar los servidores principales es la organización conocida como *Vrije Universiteit* en el país *NL*, pero no estamos interesados en una unidad organizacional en particular. Sin embargo, cada resultado devuelto debe tener el atributo *NC=Servidor principal*.

Como ya mencionamos, buscar en un servicio de directorio es en general una operación cara. Por ejemplo, para encontrar los servidores principales de la Vrije Universiteit es necesario buscar todas las entradas de cada departamento, y combinar los resultados en una sola respuesta. En otras palabras, generalmente necesitaremos acceder a varios nodos hoja de un DIT para poder obtener una respuesta. En la práctica, esto también significa que será necesario acceder a varios DSA. Por contraste, los servicios de nombre con frecuencia pueden implementarse de tal manera que una operación de búsqueda requiera acceder únicamente a un solo nodo hoja.

Toda la configuración de LDAP puede ir un paso más allá si se permite que varios árboles coexistan y estén vinculados uno con otro. Este método se sigue en el Active Directory de Microsoft, lo cual lleva a un *bosque* de dominios LDAP (Allen y Lowe-Norris, 2003). Desde luego, buscar en

una organización como ésta puede ser extremadamente complejo. Para evitar algunos problemas de escalabilidad, Active Directory generalmente supone que hay un servidor global de índices (conocido como catálogo global) en donde se puede buscar primero. El índice mostrará cuáles dominios LDAP necesitarán de una mayor búsqueda.

Aunque LDAP, por sí mismo, ya aporta la jerarquía para escalabilidad, resulta común combinar LDAP con DNS. Por ejemplo, todo árbol en LDAP necesita ser accesible en la raíz (conocido en Active Directory como controlador de dominio). A menudo la raíz es conocida con un nombre DNS el cual, a su vez, puede ser encontrado mediante un registro RSV adecuado como explicamos antes.

LDAP generalmente representa una forma estándar de soportar los nombres basados en atributos. También se han desarrollado recientemente otros servicios de directorio que siguen este método tradicional, de modo notable en el contexto de cómputo en grid y servicios web. Un ejemplo específico es el **directorio universal y la integración de descubrimientos**, o simplemente **UDDI** por sus siglas en inglés.

Estos servicios suponen una implementación en la que uno o sólo algunos nodos cooperan para mantener una sencilla base de datos distribuida. Desde un punto de vista tecnológico, realmente no hay novedad aquí. De igual manera, no hay algo realmente nuevo que informar cuando se trata de introducir nueva terminología, como puede observarse cuando se revisan los cientos de páginas de las especificaciones UDDI (Clement y cols., 2004). El esquema básico siempre es el mismo. La escalabilidad se logra haciendo que varias de estas bases de datos sean accesibles para las aplicaciones, las cuales son responsables de consultar cada base de datos en forma separada y de agregar los resultados. Principalmente para soporte al middleware.

5.4.3 Implementaciones descentralizadas

Con la llegada de los sistemas punto a punto, los investigadores también han buscado soluciones para sistemas de nombres descentralizados basados en atributos. El punto clave aquí es que los pares (*atributo, valor*) necesitan estar mapeados de manera eficiente para que la búsqueda sea también eficiente, esto es, evitar una búsqueda exhaustiva a través de todo el espacio de atributos. A continuación daremos un vistazo a las diferentes formas de establecer dicho mapeo.

Mapeo hacia tablas hash distribuidas

Consideremos primero el caso en donde los pares (*atributo*, *valor*) necesitan ser soportados por un sistema basado en DHT. Primero, asumamos que las consultas constan de una conjunción de pares con LDAP, esto es, un usuario especifica una lista de atributos, junto con el valor único que desea ver en cada atributo respectivo. La principal ventaja de este tipo de consulta es que no se requiere soportar rango alguno. Las consultas por rango pueden incrementar de manera significativa la complejidad de los pares de mapeo hacia un DHT.

Las consultas de un solo valor están soportadas en el sistema INS/Twine (Balazinska y cols., 2002). Se asume que cada entidad (referida como un recurso) está descrita por medio de atributos posiblemente organizados de manera jerárquica, tal como los que aparecen en la figura 5-24. Cada

una de las descripciones se traduce a un **árbol atributo-valor** (**AVTree**), el cual se utiliza entonces como la base de una codificación que coincide bien dentro de un sistema basado en DHT.



Figura 5-24. (a) Descripción general de un recurso. (b) Su representación como un árbol AVTree.

El punto principal es transformar los AVTrees en una colección de llaves que se puedan respaldar dentro de un sistema DHT. En este caso, a cada ruta que se origina en la raíz se le asigna un valor hash único, en donde la descripción de la ruta comienza con un vínculo (que representa un atributo), y termina ya sea con un nodo (valor) o con otro vínculo. Al tomar la figura 5-24(b) como nuestro ejemplo, consideraremos las siguientes hashes (dispersiones) de dichas rutas:

h_1: hash(tipo-libro)
h_2: hash(tipo-libro-autor)
h_3: hash(tipo-libro-autor-Tolkien)
h_4: hash(tipo-libro-título)
h_5: hash(tipo-libro-título-LOTR)
h_6: hash(género-fantasía)

Un nodo responsable del valor hash h_i mantendrá (una referencia hacia) el recurso real. En nuestro ejemplo, esto puede provocar que seis nodos almacenen información acerca de la obra *Lord of the Rings*, de Tolkien. Sin embargo, el beneficio de esta redundancia es que permitirá soportar consultas parciales. Por ejemplo, considere una consulta como "Devuelve libros escritos por Tolkien". Esta consulta se traduce a un AVTree como aparece en la figura 5-25, lo que provoca el cálculo de los siguientes hashes en el árbol:

h_1: hash(tipo-libro)
h_2: hash(tipo-libro-autor)
h_3: hash(tipo-libro-autor-Tolkien)

Estos valores se enviarán a los nodos que almacenan información acerca de los libros de Tolkien, y al menos devolverán *Lord of the Rings*. Observe que un hash como h_1 es más general y se generará con frecuencia. Este tipo de hashes se puede filtrar y dejar fuera del sistema. Más aún, no es difícil advertir que solamente se necesita evaluar los hashes más específicos. Podemos encontrar más detalles en Balzinska y colaboradores (2002).

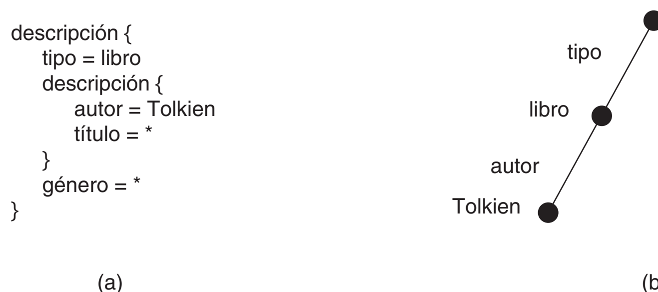


Figura 5-25. (a) Descripción del recurso de una consulta. (b) Su representación como un árbol AVTree.

Ahora demos un vistazo a otro tipo de consulta, a saber, las consultas que contienen especificaciones de rangos para valores de atributo. Por ejemplo, alguien que busca una casa por lo general desea especificar que el precio debe caer dentro de un rango específico. De nuevo, se han propuesto diversas soluciones y revisaremos algunas de ellas cuando expliquemos los sistemas de publicación/suscripción en el capítulo 13. Aquí, explicaremos una solución adoptada en el sistema de descubrimiento de recursos SWORD (Oppenheimer y cols., 2005).

En SWORD, los pares (*atributo*, *valor*) proporcionados por la descripción de un recurso se transforman primero en la llave para un DHT. Observe que estos pares siempre contienen un solo valor; solamente las consultas pueden contener rangos de valores para los atributos. Al calcular el hash, el nombre del atributo y su valor se almacenan por separado. En otras palabras, los bits específicos en la llave resultante identificarán al nombre del atributo, mientras que otros identificarán su valor. Además, la llave contendrá cierto número aleatorio de bits para garantizar la unicidad a lo largo de todas las llaves que se deben generar.

Así, el espacio de atributos se particiona de manera conveniente: si se reservan n bits para el código de los nombres de atributo, entonces se utilizarán 2^n grupos de servidores diferentes, un grupo por cada nombre de atributo. En forma similar, mediante el uso de m bits para reforzar los valores codificados, una partición adicional por grupo de servidores se puede aplicar para almacenar pares (*atributo*, *valor*) específicos.

Por cada nombre de máquina, un posible rango de su valor se particiona dentro de subrangos y se asigna un solo servidor a cada subrango. Para explicarlo, considere una descripción de recurso con dos atributos: a_1 toma valores en el rango de [1..10] y a_2 toma valores en el rango [101...200]. Asumamos que existen dos servidores para a_1: s_{11} cuida de grabar los valores de a_1 en [1..5], y s_{12} para valores en [6..10]. De manera similar, los registros del servidor s_{21} almacenan valores para a_2 en el rango [101..150] y el servidor s_{22} para valores en [151..200]. Entonces, cuando el recurso obtiene sus valores ($a_1 = 7$, $a_2 = 175$), los servidores s_{12} y s_{22} tendrán que ser informados.

La ventaja de este esquema es que las consultas de rango se pueden soportar fácilmente. Cuando se lanza una consulta para que devuelva recursos que contengan valores en a_2 que oscilen entre 165 y 189, la consulta se puede enviar hacia el servidor s_{22} el cual puede entonces devolver el valor de los recursos que coinciden con el rango de la consulta. Sin embargo, la desventaja es que las actualizaciones necesitan enviarse a múltiples servidores. Más aún, no queda claro de inmediato lo

bien que el balanceo de cargas se realiza entre los distintos servidores. En especial, cuando cierto rango de consultas resulta ser muy popular, los servidores específicos recibirán una parte importante de todas las consultas. La manera en que se puede atacar este problema de balanceo en sistemas basados en DHT se explica en Bharambe y colaboradores (2004).

Redes sobrepuestas semánticas

Las actuales implementaciones descentralizadas de nombres basados en atributos muestran un incremento en el grado de autonomía de los distintos nodos. El sistema es menos sensible a los nodos que se unen o abandonan en comparación, por ejemplo, con los sistemas basados en LDAP. Este grado de autonomía se incrementa adicionalmente cuando los nodos contienen descripciones de recursos que están ahí para ser descubiertos por otros. En otras palabras, no existe un esquema determinístico *a priori* mediante el cual los pares (*atributo*, *valor*) puedan esparcirse a lo largo de una colección de nodos.

No tener dicho esquema obliga a los nodos a descubrir en dónde se encuentran los recursos solicitados. Este descubrimiento es típico de las redes sobrepuestas semánticas, las cuales explicamos en el capítulo 2. Con el objeto de volver la búsqueda más eficiente, es importante que un nodo tenga referencias hacia otros nodos que puedan responder certeramente a esas consultas. Si asumimos que las consultas originadas desde el nodo *P* se relacionan fuertemente con los recursos que tiene *P*, entonces buscaremos proveer a *P* con una colección de vínculos hacia sus vecinos *semánticamente próximos*. Recuerde que una lista de tal tipo es también conocida como **vista parcial**. La proximidad semántica se puede definir de diferentes maneras, pero se evapora al intentar mantener el rastro de los nodos con recursos similares. Los nodos y estos vínculos formarán entonces lo que conocemos como una **red sobrepuesta semántica**.

Un método común para las redes sobrepuestas semánticas es asumir que existen valores comunes en la metainformación almacenada en cada nodo. En otras palabras, los recursos almacenados en cada nodo son descritos mediante el uso de la misma colección de atributos, o, de manera más precisa, el mismo esquema de datos (Crespo y García-Molina, 2003). Tener tal esquema permitirá la definición de funciones de similitud entre los nodos. Cada nodo contendrá solamente los vínculos hacia los *K* vecinos más similares y consultará primero dichos nodos al buscar datos específicos. Observe que este método tiene sentido solamente si podemos asumir, de manera general, que una consulta iniciada en un nodo se relaciona con el contenido almacenado en dicho nodo.

Por desgracia, generalmente es un error asumir la similitud en los esquemas de datos. En la práctica, la metainformación con respecto a los recursos es altamente inconsistente a lo largo de diferentes nodos, y alcanzar el consenso sobre el qué y el cómo describir los recursos es casi imposible. Por esta razón, a menudo las redes sobrepuestas semánticas necesitarán encontrar formas diferentes para definir la similitud.

Un método es olvidarse de los atributos en conjunto y considerar solamente descriptores muy sencillos, tales como los nombres de archivo. La construcción pasiva de una sobreposición se puede realizar siguiendo aquellos nodos que respondan de manera positiva a la búsqueda de archivos. Por ejemplo, Sripanidkulchai y colaboradores (2003) envían primero una consulta a un vecino semántico del nodo, pero si el archivo solicitado no existe, se realiza una transmisión (limitada). Por supuesto, dicha transmisión puede provocar una actualización de la lista de vecinos semánticos. Como

nota, es interesante observar que si un nodo solicita a sus vecinos semánticos el reenvío de una consulta hacia *sus* vecinos semánticos, el efecto es mínimo (Handrukande y cols., 2004). Este fenómeno se puede explicar mediante lo que conocemos como **efecto del mundo pequeño**, el cual esencialmente establece que los amigos de Alice son también amigos entre sí (Watts, 1999).

Un método más proactivo para la construcción de una lista de vecinos semánticos lo proponen Voulgaris y Van Steen (2005), quienes utilizan una **función de proximidad semántica** definida en las listas de archivos FL_p y FL_Q de dos nodos P y Q, respectivamente. Esta función simplemente cuenta el número de archivos comunes en FL_p y FL_Q. La meta es entonces optimizar la función de proximidad al permitir que un nodo mantenga una lista de sólo aquellos vecinos que tienen el mayor número de archivos en común con él.

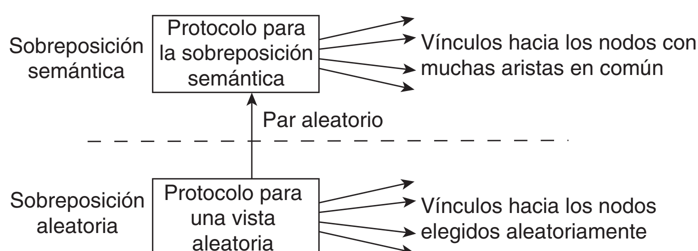


Figura 5-26. Mantenimiento de una sobreposición semántica en una conversación.

Hasta este punto, un esquema de conversación de dos capas se muestra como aparece en la figura 5-26. La capa del fondo consta de un protocolo epidémico que ayuda al mantenimiento de una vista parcial de nodos seleccionados de manera uniforme al azar. Existen diferentes formas de lograr esto, como explicamos en el capítulo 2 [vea además Jelasity y cols., (2005a)]. La capa superior mantiene una lista de los vecinos semánticos más cercanos a lo largo de la conversación. Para iniciar el intercambio, un nodo P puede seleccionar al azar a un vecino Q desde su lista actual, pero el truco es permitir que P envíe solamente aquellas entradas que son semánticamente más cercanas a Q. En cambio, cuando P recibe entradas desde Q, mantendrá una vista parcial que consta solamente de los nodos más cercanos semánticamente. Como resultado, las vistas parciales que se mantienen mediante la capa superior convergirán rápidamente hacia lo óptimo.

Como queda claro hasta este punto, las redes sobrepuestas semánticas se relacionan íntimamente con la búsqueda descentralizada. En Risson y Moors (2006) se encuentra una explicación extensa acerca de la búsqueda en todos los tipos de sistemas punto a punto.

5.5 RESUMEN

Los nombres se utilizan para hacer referencia a las entidades. De manera esencial, existen tres tipos de nombres. Una dirección es el nombre de un punto de acceso asociado con una entidad, también llamada dirección de una entidad. Un identificador es otro tipo de nombre; y tiene tres propiedades:

a cada entidad se hace referencia empleando exactamente un identificador, un identificador hace referencia solamente a una entidad, y nunca se asigna a otra entidad. Finalmente, los nombres amigables para el usuario están destinados al uso de las personas y, como tales, son representados mediante cadenas de caracteres. Dados estos tipos, hacemos una distinción entre nombres planos, nombres estructurados, y nombres basados en atributos.

Los sistemas para nombres planos requieren en esencia resolver un identificador hacia la dirección de su entidad asociada. Esta ubicación de una entidad se puede llevar a cabo de diferentes maneras. El primer método es usar transmisión o multitransmisión. El identificador de la entidad se transmite a cada proceso localizado en el sistema distribuido. El proceso que ofrece el punto de acceso para la entidad responde a proporcionar una dirección para dicho punto de acceso. Desde luego, este método tiene escalabilidad limitada.

Un segundo método es el uso de apuntadores hacia adelante. Cada vez que una entidad se mueve a la siguiente ubicación, deja atrás un apuntador que indica en dónde estará a continuación. Localizar la entidad requiere recorrer la ruta de los apuntadores hacia adelante. Para evitar grandes cadenas de apuntadores, es importante reducir periódicamente las cadenas.

Un tercer método es asignar un origen para una entidad. Cada vez que una entidad se mueve hacia otra asignación, le informa al origen en dónde se encuentra. La ubicación de una entidad se lleva a cabo preguntando primero a su origen por la ubicación actual.

Una cuarto método es organizar todos los nodos dentro de un sistema punto a punto estructurado, y asignar de manera sistemática los nodos a las entidades tomando en cuenta sus respectivos identificadores. Mediante la subdivisión subsecuente de un algoritmo de ruteo por medio del cual las peticiones de búsqueda se mueven hacia el nodo responsable para una identidad dada, es posible volver más eficiente y robusta la resolución de nombres.

Un quinto método es construir un árbol jerárquico de búsqueda. La red está dividida en dominios no traslapados. Los dominios se pueden agrupar dentro de dominios de mayor nivel (no traslapados), y así sucesivamente. Existe un solo dominio del más alto nivel que cubre toda la red. Cada dominio ubicado en cada nivel contiene un nodo directorio asociado. Si una entidad se localiza en un dominio D, el nodo directorio del dominio del siguiente nivel más alto tendrá un apuntador hacia D. Un nodo directorio de más bajo nivel almacena la dirección de la entidad. El nodo directorio de nivel más alto sabe acerca de todas las entidades.

Los nombres estructurados se organizan fácilmente dentro de un espacio de nombre. Un espacio de nombre se puede representar mediante un grafo de nombres donde el nodo representa una entidad con nombre y la etiqueta del borde representa el nombre con el cual se conoce a esta entidad. Un nodo que tiene muchas aristas salientes representa una colección de entidades y también se le conoce como nodo de contexto o nodo directorio. Con frecuencia, los grafos de nombres a gran escala se organizan como grafos directos con raíz y sin ciclo.

Los grafos de nombres son adecuados para organizar nombres amigables con el usuario de manera estructurada. Se puede hacer referencia a una entidad mediante el nombre de ruta. La resolución de nombres es el proceso de recorrer los grafos de nombres mediante la revisión de los componentes del nombre de ruta, un componente a la vez. Un grafo de nombres de gran escala se implementa mediante la distribución de sus nodos a lo largo de múltiples servidores. Al resolver el nombre de ruta empleando el recorrido del grafo de nombres, la resolución de nombres continúa en el siguiente servidor de nombre tan pronto como se alcanza el nodo implementado por dicho servidor.

Más problemáticos son los esquemas basados en atributos en los cuales las entidades se describen mediante una colección de pares (*atributo*, *valor*). Las consultas también se formulan como dichos pares, requiriendo esencialmente una búsqueda exhaustiva a través de todos los descriptores. Dicha búsqueda solamente es posible cuando los descriptores se almacenan en una sola base de datos. Si embargo, se han desarrollado soluciones alternativas mediante las cuales los pares se mapean dentro de sistemas basados en DHT, lo cual provoca fundamentalmente una distribución de una colección de descriptores de entidades.

Relacionado con los nombres basados en atributos está el proceso de reemplazar de manera gradual la resolución de nombres mediante técnicas distribuidas de búsqueda. Este método se lleva a cabo en redes sobrepuestas semánticas, donde los nodos mantienen una lista local de otros nodos que tienen contenido semántico similar. Estas listas semánticas permiten llevar a cabo la búsqueda eficiente por medio de la cual se solicitan primero los vecinos inmediatos, y sólo cuando esto no tiene éxito se lleva a cabo la instalación de la difusión (limitada).

PROBLEMAS

1. Escriba un ejemplo en donde la dirección de una entidad *E* necesita resolverse dentro de otra dirección para acceder en realidad a *E*.

2. ¿Considera usted que una dirección URL como *http://www.acme.org/index.html* es una dirección independiente? ¿Y que tal *http://www.acme.nl/index.html*?

3. Escriba algunos ejemplos de identificadores verdaderos.

4. ¿Se permite a un identificador contener información acerca de una entidad a la que hace referencia?

5. ¿Bosqueje una implementación eficiente para identificadores globalmente únicos?

6. Considere el sistema de cuerdas que aparece en la figura 5-4 y asuma que el nodo 7 acaba de conectarse a la red. ¿Defina cuál sería su tabla finger y explique si habría cambio alguno en otras tablas finger?

7. Considere el sistema de cuerdas basado en DTH para el cual se reservan *k* bits de un identificador de *m* bits para asignar los superpares. Si los identificadores se asignan de manera aleatoria, ¿cuántos superpares podemos esperar tener para un sistema de *N* nodos?

8. Si insertamos un nodo dentro del sistema de cuerdas, ¿necesitaremos actualizar de manera instantánea todas las tablas finger?

9. ¿Cuál es una de las mayores desventajas de las búsquedas recursivas al resolver una llave dentro de un sistema basado en DHT?

10. Existe una forma especial para localizar una entidad, llamada anycasting, mediante la cual identificamos un servicio usando una dirección IP (por ejemplo, vea RFC 1546). Esta dirección envía una petición a una dirección anycast, devuelve la respuesta desde el servidor, e implementa el servicio

identificado por la dirección anycast. Bosqueje la implementación de un servicio anycast basado en la ubicación jerárquica del servicio descrito en la sección 5.2.4.

11. Si se considera que el método de dos hilos basado en el origen es una especialización de un servicio basado en el servicio de ubicación jerárquica, ¿en dónde se encuentra la raíz?

12. Suponga que sabemos que una entidad móvil específica nunca se moverá fuera del dominio *D*, y si lo hace, podemos esperar que regrese pronto. ¿Cómo podemos utilizar esta información para acelerar la operación de búsqueda dentro de un servicio de ubicación jerárquica?

13. En un servicio de ubicación jerárquica con profundidad *k*, ¿cuántos registros de ubicación se deben actualizar como máximo cuando una entidad móvil modifica su ubicación?

14. Considere una entidad que se mueve desde una ubicación *A* hacia *B*, mientras pasa por distintas ubicaciones intermedias hasta donde residirá durante un corto periodo. Cuando llega al punto *B*, se estabiliza por un tiempo. Modificar una dirección en un servicio de ubicación jerárquica pudiera tomar un tiempo relativamente largo para completarse, y por tanto pudiera evitarse al visitar una ubicación intermedia. ¿Cómo se puede ubicar la entidad en una ubicación intermedia?

15. El nodo raíz en los servicios de ubicación jerárquica pudiera ser potencialmente un cuello de botella. ¿Cómo se puede evitar este problema de manera efectiva?

16. Escriba un ejemplo de la forma en que pudiera funcionar un mecanismo para el cierre de una URL.

17. Explique la diferencia entre un vínculo duro y un vínculo suave en los sistemas UNIX. ¿Existen cosas que hacemos con un vínculo duro que no se puedan hacer con un vínculo suave o viceversa?

18. Por lo general, los servidores de alto nivel en los DNS, esto es, los servidores de nombre que implementan nodos en el espacio de nombre del DNS cerrados en la raíz, no soportan la resolución recursiva de nombres. ¿Podemos esperar un aumento significativo del rendimiento si lo hacen?

19. Explique cómo podemos utilizar un DNS para implementar un método basado en el origen y localizar los servidores móviles.

20. ¿Cómo se bloquea un punto de montaje en la mayoría de los sistemas UNIX?

21. Considere un sistema de archivos distribuidos que utiliza espacios de nombre punto a punto. En otras palabras, cada usuario tiene su propio espacio de nombre. ¿Podemos utilizar dichos espacios para compartir recursos entre dos usuarios diferentes?

22. Considere un DNS. Para hacer referencia a un nodo *N* en un subdominio implementado como una zona diferente al dominio actual, es necesario especificar un servidor de nombre para esa zona. ¿Será necesario incluir siempre un registro de recurso para la dirección de dicho servidor, o en ocasiones es suficiente con proporcionar sólo el nombre de dominio?

23. Contar los archivos comunes es una manera ingenua de definir la proximidad semántica. Asuma que usted construirá redes con una capa semántica basada en documentos de texto, ¿en qué otra función de proximidad semántica puede pensar?

24. (**Asignación para el laboratorio.**) Configure su propio servidor DNS. Instale BIND tanto en una máquina Windows como en una máquina UNIX y configúrelas para implementar unos cuantos nombres sencillos. Evalúe su configuración mediante el uso de herramientas tales como el Domain

Information Groper (DIG). Asegúrese de que su base de datos DNS incluye registros para servidores de nombre, servidores de correo, y servidores estándar. Observe que si ejecuta BIND en una máquina cuyo nombre de servidor es *NOMBRESERVIDOR*, debiera ser capaz de resolver nombres de la forma *RECURSO-NOMBRE.NOMBRESERVIDOR*.

6

SINCRONIZACIÓN

En los capítulos anteriores revisamos los procesos y la comunicación entre procesos. Si bien la comunicación resulta importante, no lo es todo. La manera en que los procesos cooperan y se sincronizan entre sí está muy relacionada. La cooperación es parcialmente soportada mediante la asignación de nombres, la cual permite a los procesos compartir al menos recursos, o entidades en general.

En este capítulo nos concentraremos principalmente en estudiar cómo pueden sincronizarse los procesos. Por ejemplo, es importante que varios procesos no accedan simultáneamente a un recurso compartido, digamos a una impresora, sino que cooperen para garantizar a cada uno el acceso exclusivo temporal al recurso. Otro ejemplo es que ocasionalmente varios procesos pueden acordar el orden de los eventos, tal como que si el mensaje *m1* del proceso *P* se envió antes o después del mensaje *m2* del proceso *Q*.

Como veremos, la sincronización en sistemas distribuidos es con frecuencia mucho más difícil comparada con la sincronización en sistemas de un procesador o de multiprocesadores. Los problemas y soluciones que explicamos en este capítulo son, por su naturaleza, más generales y ocurren en muchas situaciones diferentes en sistemas distribuidos.

Comenzamos con el análisis de un asunto de sincronización basado en tiempo real, seguido de la sincronización en la que sólo importan cuestiones de ordenamiento relativo en lugar del ordenamiento en tiempo absoluto.

En muchos casos, es importante que un grupo de procesos pueda designar a un proceso como coordinador, lo cual puede hacerse mediante algoritmos de elección. En una sección aparte explicaremos varios algoritmos de elección.

Los algoritmos distribuidos vienen en todos los tipos y sabores, y se han desarrollado para diferentes clases de sistemas distribuidos. En Andrews (2000) y Guerraoui y Rodrigues (2006) pueden encontrarse varios ejemplos. En los libros de texto de Attiya y Welch (2004), Lynch (1996), y (Tel, 2000), podemos encontrar métodos más formales y diversos algoritmos.

6.1 SINCRONIZACIÓN DEL RELOJ

En un sistema centralizado, el tiempo no es ambiguo. Cuando un proceso quiere saber la hora, realiza una llamada de sistema y el núcleo se la dice. Si un proceso *A* pregunta la hora, y después un proceso *B* pregunta la hora, el valor que obtiene *B* será mayor que (o probablemente igual que) el valor que obtuvo *A*; con certeza no será menor. En un sistema distribuido, lograr un acuerdo con respecto al tiempo no es algo trivial.

Por un momento, y a manera de ejemplo, pensemos sólo en las implicaciones de carecer de un tiempo global en el programa *make* de UNIX. En general, en UNIX, los programas grandes se dividen en varios archivos fuente, de modo tal que un cambio en un archivo fuente sólo necesita compilar un archivo, y no todos. Si un programa consta de 100 archivos, el hecho de no tener que recompilar todo, debido a que un archivo se modificó, incrementa enormemente la velocidad a la que los programadores pueden trabajar.

La forma en que *make* funciona es sencilla. Cuando el programador ha terminado de modificar todos los archivos fuente, ejecuta *make*, el cual analiza las fechas en que todos los archivos fuente y objeto se modificaron por última vez. Si el archivo fuente *input.c* tiene la hora 2151, y el archivo objeto correspondiente *input.o* tiene la hora 2150, *make* sabe que *input.c* ha cambiado desde que *input.o* se creó, y por tanto *input.c* debe recompilarse. Por otra parte, si *output.c* tiene la hora 2144 y *output.o* tiene la hora 2145, no es necesario recompilar. Así, *make* revisa todos los archivos fuente para ver cuál necesita recompilarse, y llama al compilador para que lo haga.

Ahora imaginemos lo que podría pasar en un sistema distribuido en el que no existiera un acuerdo global con respecto al tiempo. Supongamos que *output.o* tiene la hora 2144 como indicamos arriba, y que poco después *output.c* se modifica pero se le asigna la hora 2143 debido a que el reloj de su máquina está ligeramente retrasado, como ilustra la figura 6-1. *Make* no llamará al compilador. El programa binario ejecutable resultante entonces contendrá una mezcla de archivos objeto proveniente de las fuentes anteriores y de las fuentes nuevas; el programa probablemente fallará y el programador enloquecerá intentando comprender lo que está mal en el código.

Hay muchos más ejemplos acerca de procesos en los que se necesita precisión en el tiempo. El ejemplo anterior, puede replantearse fácilmente para archivos de registro de tiempo en general. Además, consideremos aplicaciones tales como correduría financiera, auditoría de seguridad, y detección de colaboración y resultará evidente la importancia de la sincronización. Debido a que el tiempo es esencial en la forma de pensar de la gente y el efecto de no sincronizar los relojes puede volverse catastrófico, es conveniente que iniciemos nuestro estudio sobre la sincronización con una sencilla pregunta: ¿es posible sincronizar todos los relojes de un sistema distribuido? La respuesta es sorprendentemente complicada.

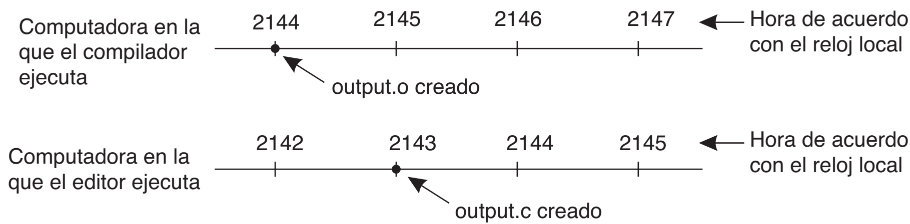


Figura 6-1. Cuando cada máquina tiene su propio reloj, es posible que a un evento que ocurrió después de otro se le asigne una hora anterior.

6.1.1 Relojes físicos

Casi todas las computadoras tienen un circuito para dar seguimiento al tiempo. A pesar del amplio uso de la palabra “reloj” para hacer referencia a estos dispositivos, en realidad no son relojes en el sentido usual. Tal vez **cronómetro** sea una mejor palabra con la cual designarlos. Un cronómetro de computadora, en general, es un cristal de cuarzo mecanizado con precisión. Cuando este dispositivo se mantiene sujeto a tensión, los cristales de cuarzo oscilan en una frecuencia bien definida que depende del tipo de cristal, de la forma de su corte, y de la cantidad de tensión. Hay dos registros asociados con cada cristal, un **contador** y un **registro de mantenedor**. Cada oscilación del cristal disminuye el contador en uno. Cuando el contador llega a cero, se genera una interrupción y el contador se reinicia a partir del registro de mantenedor. De esta manera, es posible programar un cronómetro para generar una interrupción 60 veces por segundo, o a cualquier otra frecuencia deseada. A cada interrupción se le conoce como **marca de reloj**.

Cuando un sistema se inicia, generalmente solicita al usuario la fecha y la hora, las cuales se convierten en el número de marcas posteriores a una fecha de inicio conocida y se almacena en memoria. La mayoría de las computadoras tiene una pila especial que respalda la RAM CMOS de tal modo que no es necesario introducir la fecha ni la hora en inicios posteriores. En cada marca del reloj, el procedimiento de servicio de interrupción agrega uno a la hora almacenada en memoria. De esta manera, el reloj (software) se mantiene actualizado.

Con una sola computadora y un solo reloj, no importa mucho si este reloj está desfasado por una pequeña cantidad. Debido a que todos los procesos de la máquina utilizan el mismo reloj, serán internamente consistentes. Por ejemplo, si el archivo *input.c* tiene la hora 2151 y el archivo *input.o* tiene la hora 2150, *make* recompilará el archivo fuente, incluso si el reloj está desfasado por 2 y las horas reales son 2153 y 2152, respectivamente. Lo que importa en realidad son los tiempos relativos.

Tan pronto como se introducen varias CPU, cada una con su propio reloj, la situación cambia radicalmente. Aunque la frecuencia con la que oscila un cristal es en general bastante estable, resulta imposible garantizar que los cristales de las diferentes computadores funcionen exactamente con la misma frecuencia. En la práctica, cuando un sistema tiene *n* computadoras, los *n* cristales funcionarán a velocidades ligeramente diferentes, lo cual ocasiona que los relojes (software) se salgan gradualmente de sincronía y arrojen diferentes valores cuando se leen. Esta diferencia en valores se

conoce como **distorsión de reloj**. Como una consecuencia de este desajuste de reloj, los programas que esperan que la hora asociada con un archivo, objeto, proceso o mensaje sea correcta e independiente de la máquina en la que se generaron (es decir, del reloj utilizado) pueden fallar, tal como vimos en el anterior ejemplo de *make*.

En algunos sistemas (por ejemplo, sistemas de tiempo real), la hora del reloj real es importante. Bajo estas circunstancias, los relojes físicos externos son necesarios. Por razones de eficiencia y redundancia, generalmente se considera deseable tener varios relojes, lo cual genera dos problemas: (1) ¿cómo los sincronizamos con relojes reales?, y (2) ¿cómo los sincronizamos entre sí?

Antes de responder estas preguntas, desviémonos un poco para ver cómo se mide en realidad el tiempo. No es en absoluto tan simple como podría pensarse, especialmente cuando se requiere alta precisión. Desde que se inventaron los relojes mecánicos en el siglo XVII, el tiempo se ha medido astronómicamente. Cada día el Sol parece surgir por el Este, después sube a su máxima altura en el cielo, y finalmente desaparece por el Oeste. Al hecho de que el Sol alcance su punto aparentemente más alto en el cielo, se le conoce como **trayectoria solar**. Este hecho ocurre aproximadamente cada mediodía. El intervalo entre dos trayectorias consecutivas se conoce como **día solar**. Debido a que hay 24 horas en un día, cada una con 3 600 segundos, el **segundo solar** se define exactamente como 1/86 400 de un día solar. La geometría del cálculo de la media de un día solar aparece en la figura 6-2.

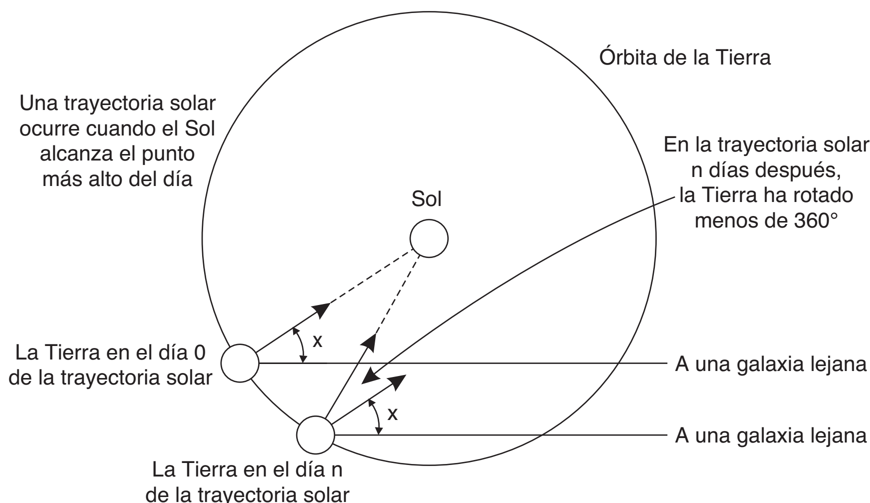


Figura 6-2. Cálculo de la media de un día solar.

En la década de 1940 se estableció que el periodo de rotación de la Tierra no es constante. La velocidad de la Tierra está disminuyendo debido a la fricción de las mareas y a la resistencia atmosférica. Con base en estudios de patrones de crecimiento de corales antiguos, ahora los geólogos creen que hace 300 millones de años había 400 días por año. No se cree que la duración del año (el tiempo para realizar un viaje alrededor del Sol) haya cambiado, sino que simplemente el día se ha vuelto más largo. Además de esta tendencia de larga duración, también ocurren variaciones de

corta duración a lo largo del día, probablemente ocasionadas por la gran turbulencia de la lava del centro de la Tierra. Estas revelaciones llevaron a los astrónomos a calcular la duración del día midiendo muchos días y tomando el promedio antes de dividirlo entre 86400. La cantidad resultante fue llamada **segundo medio solar**.

Con la invención del reloj atómico en 1948, se hizo posible medir el tiempo de manera más precisa e independiente de los movimientos de la Tierra, a través del conteo de las transiciones del átomo de cesio 133. Los físicos asumieron el trabajo de los astrónomos de medir el tiempo, y definieron al segundo como el tiempo que necesita el átomo de cesio 133 para efectuar exactamente 9192631770 transiciones. La elección de 9192631770 se hizo para igualar el segundo atómico con el segundo medio solar en el año de su introducción. En la actualidad, diversos laboratorios del mundo tienen relojes de cesio 133. De manera periódica, cada laboratorio le indica al Bureau International de l'Heure (BIH) de París cuántas veces ha hecho marca su reloj. El BIH promedia estas marcas para producir el **Tiempo Atómico Internacional**, lo cual se abrevia como **TAI**. Así, el TAI es exactamente el número medio de marcas de los relojes de cesio 133, desde la medianoche del 1 de enero de 1958 (el comienzo del tiempo), dividido entre 9192631770.

Aunque el TAI es muy estable y está disponible para cualquiera que desee encarar la cuestión de comprar un reloj de cesio, hay un serio problema con él; 86400 segundos TAI son ahora aproximadamente 3 ms menos que un día medio solar (debido a que el día medio solar se está volviendo más largo todo el tiempo). Utilizar el TAI para contar el tiempo significaría que a través del curso de los años, el mediodía llegaría más y y más temprano, hasta que en algún momento llegara en las primeras horas de la mañana. La gente podría advertir esto y tendríamos el mismo tipo de situación que ocurrió en 1582 cuando el Papa Gregorio XIII decretó la eliminación de 10 días del calendario. Este evento ocasionó disturbios en las calles porque los caseros demandaron la renta completa del mes y los banqueros sus intereses, mientras que los patrones se negaron a pagar a los empleados por 10 días que no trabajaron, por mencionar sólo algunos conflictos. Los países de religión protestante, como una cuestión de principios, se rehusaron a tener algo que ver con el decreto papal, y no aceptaron el calendario gregoriano durante 170 años.



Figura 6-3. Los segundos TAI son de longitud constante, a diferencia de los segundos solares. Los segundos vacíos se introducen cuando es necesario mantenerlos en fase con el Sol.

El BIH resuelve el problema introduciendo **segundos vacíos** siempre que la discrepancia entre el TAI y el tiempo solar alcanza los 800 ms. El uso de segundos vacíos aparece en la figura 6-3.

Esta corrección da lugar a un sistema de tiempo basado en segundos TAI constantes, pero que permanecen en fase con el movimiento aparente del Sol. A este sistema se le denomina **Tiempo Universal Coordinado** y se abrevia como **UTC** (por sus siglas en inglés). El UTC es la base de todo conteo de tiempo civil moderno; prácticamente ha reemplazado el viejo estándar, Tiempo Medio de Greenwich, que es el tiempo astronómico.

La mayoría de las empresas de energía eléctrica sincronizan el tiempo de sus relojes de 60 o 50 Hz con el UTC, por lo que cuando el BIH anuncia un segundo vacío, estas empresas elevan su frecuencia a 61 o 51 Hz por 60 o 50 segundos para adelantar todos los relojes de su área de distribución. Debido a que un segundo es un intervalo digno de atención para una computadora, un sistema operativo que necesite mantener el tiempo preciso durante cierto periodo de años debe tener un software especial para contar los segundos vacíos conforme sean anunciados (a menos que utilice la línea de energía para el tiempo, lo cual resulta en general demasiado rudo). El número total de segundos vacíos introducidos al UTC hasta el momento es de aproximadamente 30.

Para proporcionar el UTC a la gente que necesita el tiempo exacto, el National Institute of Standard Time (NIST) opera una estación de radio de onda corta con las siglas de llamado WWV desde Fort Collins, Colorado. WWV emite un pulso corto al inicio de cada segundo UTC. La precisión de WWV es de aproximadamente ± 1 ms, pero debido a las fluctuaciones atmosféricas aleatorias que pueden afectar la longitud de la ruta de la señal, en la práctica la precisión no es mejor que ± 10 ms. En Inglaterra, la estación MSF, que opera desde Rugby, Warwickshire, proporciona un servicio similar al de las estaciones de otros países.

Varios satélites terrestres también ofrecen un servicio UTC. El Geostationary Environment Operational Satellite puede proporcionar una precisión UTC de 0.5 ms, y algunos otros satélites lo hacen incluso mejor.

Utilizar ya sea radio de onda corta o servicios satelitales requiere un conocimiento preciso de la posición relativa del emisor y del receptor, para compensar el retraso en la propagación de la señal. Los receptores de radio para WWV, GEOS, y otras fuentes UTC ya están comercialmente disponibles.

6.1.2 Sistema de posicionamiento global

Como un paso hacia la solución de problemas de sincronizado de relojes, primero consideraremos un problema relacionado, a saber, cómo determinar nuestra posición geográfica en cualquier parte del planeta. Este problema de posicionamiento se resuelve por sí mismo a través de un sistema distribuido altamente específico y dedicado llamado **GPS** (por sus siglas en inglés), y que significa **sistema de posicionamiento global**. El GPS es un sistema distribuido basado en un satélite puesto en órbita en 1978. Aunque ha sido utilizado principalmente en aplicaciones militares, en años recientes ha encontrado su camino para muchas aplicaciones civiles, principalmente en la navegación. Sin embargo, existen muchos más campos de aplicación. Por ejemplo, los teléfonos GPS ahora permiten a quienes llaman rastrear la posición del otro, una característica que puede ser extremadamente útil cuando se está perdido o en problemas. Este principio puede aplicarse fácilmente para rastrear otras cosas, incluso mascotas, niños, automóviles, naves, etc. Zogg (2002) presenta una excelente cobertura sobre el GPS.

El GPS utiliza 29 satélites que circulan cada uno en una órbita situada a una altura aproximada de 20 000 km. Cada satélite tiene hasta cuatro relojes atómicos que son calibrados regularmente desde estaciones especiales ubicadas en la Tierra. Un satélite transmite su posición de manera continua, y registra el tiempo de cada mensaje con su tiempo local. Esta transmisión permite a cada receptor localizado en la Tierra calcular exactamente su propia posición, empleando en principio sólo tres satélites. Para explicar esto, primero supongamos que todos los relojes, incluyendo el del receptor, están sincronizados.

Para calcular una posición, primero consideremos el caso bidimensional que muestra la figura 6-4, donde aparecen dos satélites, junto con los círculos que representan puntos localizados a la misma distancia de su respectivo satélite. El eje *y* representa la altura, mientras que el eje *x* representa una línea recta situada a lo largo de la superficie terrestre al nivel del mar. Si ignoramos el punto más alto, vemos que la intersección de los dos círculos es un punto único (en este caso, tal vez algún lugar de una montaña).

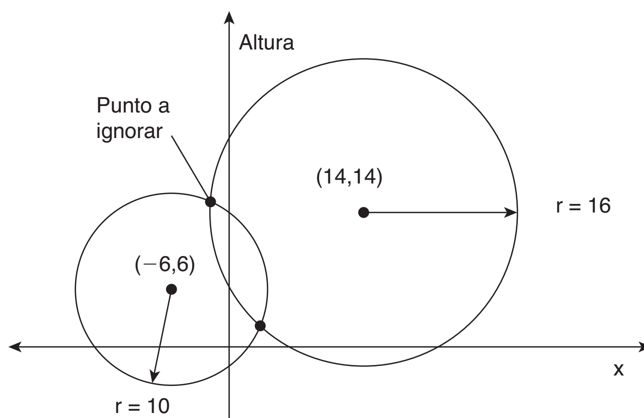


Figura 6-4. Cálculo de una posición en un espacio bidimensional.

El principio de intersecar círculos puede ampliarse a tres dimensiones, lo cual significa que necesitamos tres satélites para determinar la longitud, la latitud, y la altitud de un receptor ubicado en la Tierra. Este posicionamiento es mucho muy directo, pero las cosas se complican cuando ya no podemos suponer que todos los relojes están perfectamente sincronizados.

Existen dos hechos reales muy importantes que debemos tomar en cuenta:

1. Pasa un rato antes de que la información sobre la posición del satélite llegue al receptor.

2. El reloj del receptor generalmente no está en sincronía con el del satélite.

Supongamos que el registro del tiempo de un satélite es totalmente exacto. Denotemos con Δ_r la desviación del reloj del receptor con respecto al tiempo real. Cuando se recibe un mensaje desde el

satélite i con registro de tiempo T_i, entonces el retraso medido por el receptor, Δ_i, consiste en dos componentes: el retraso real, junto con su propia desviación:

$$\Delta_i = (T_{ahora} - T_i) + \Delta_r$$

Dado que las señales viajan a la velocidad de la luz, c, la distancia medida desde el satélite resulta ser claramente $c\Delta_i$. Con

$$d_i = c(T_{ahora} - T_i)$$

como la distancia real entre el receptor y el satélite, la distancia medida puede reescribirse como $d_i + c\Delta_r$. La distancia real simplemente se calcula como:

$$d_i = \sqrt{(x_i - x_r)^2 + (y_i - y_r)^2 + (z_i - z_r)^2}$$

donde x_i, y_i y z_i denotan las coordenadas del satélite i. Lo que vemos ahora es que si tenemos cuatro satélites obtendremos cuatro ecuaciones y cuatro incógnitas, ello nos permite resolver las coordenadas x_r, y_r y z_r para el receptor, pero también Δ_r. En otras palabras, una medición GPS proporcionará además un conteo del tiempo. Más adelante en este capítulo retomaremos el cómo determinar posiciones mediante un método similar.

Hasta el momento hemos asumido que las mediciones son perfectamente exactas. Por supuesto, no lo son. Por una parte, el GPS no considera segundos vacíos. En otras palabras, existe una desviación sistemática del UTC, la cual a partir del 1 de enero de 2006 es de 14 segundos. Tal error puede compensarse fácilmente en el software. Sin embargo, existen muchas otras fuentes de error, comenzando con el hecho de que los relojes atómicos satelitales no siempre se encuentran en perfecta sincronía, la posición de un satélite no se conoce con precisión, el reloj del receptor tiene una exactitud finita, la velocidad de propagación de la señal no es constante (la velocidad de las señales disminuye cuando, por ejemplo, entran a la ionosfera), etc. Más aún, todos sabemos que la Tierra no es una esfera perfecta, lo que nos lleva a necesitar más correcciones.

En general, el cálculo de una posición exacta va más allá de suposiciones triviales, y requiere considerar muchos detalles engorrosos. Sin embargo, incluso con receptores GPS relativamente baratos, el posicionamiento puede ser preciso en un rango de 1-5 metros. Además, los receptores profesionales (que pueden conectarse fácilmente a una red de computadoras) tienen un conocido error de aproximadamente 20-35 nanosegundos. De nuevo, hacemos referencia a la excelente visión de Zogg (2002) como un primer paso para quienes deseen conocer más detalles.

6.1.3 Algoritmos de sincronización de relojes

Si una máquina tiene un receptor WWV, el objetivo es mantener todas las demás máquinas sincronizadas con tal receptor. Si ninguna máquina tiene un receptor WWV, cada máquina da seguimiento a su propio tiempo, y el objetivo es mantenerlas juntas en tanto sea posible. Se han propuesto muchos algoritmos para realizar esta sincronización. Ramanathan y colaboradores (1990) presentan una investigación.

Todos los algoritmos tienen como base el mismo modelo del sistema. Se supone que cada máquina tiene un cronómetro que ocasiona una interrupción H veces por segundo. Cuando este cronómetro se apaga, el manipulador de interrupciones agrega 1 al reloj de software, el cual da seguimiento al número de marcas (interrupciones) a partir de algún momento pasado acordado. Llamemos C al valor de este reloj. Más específicamente, cuando el tiempo UTC es t, el valor del reloj de la máquina p es $C_p(t)$. En un mundo perfecto, tendríamos $C_p(t) = t$ para toda p y toda t. En otras palabras, $C_p'(t) = dC/dt$ idealmente debería ser 1. $C_p'(t)$ se conoce como la **frecuencia** del reloj de p en el tiempo t. La **distorsión del reloj** se define como $C_p'(t) - 1$, y denota cuánto difiere la frecuencia de la de un reloj perfecto. La **compensación** relativa a un tiempo específico t es $C_p(t) - t$.

Los cronómetros reales no interrumpen exactamente H veces por segundo. En teoría, un cronómetro con $H = 60$ debe generar 216 000 marcas por hora. En la práctica, el error relativo obtenible con chips cronómetros modernos es de alrededor de 10^{-5}, lo cual significa que una máquina en particular puede obtener un valor situado en el rango de 215 998 a 216 002 marcas por hora. De manera más precisa, si existe una constante ρ tal que

$$1 - \rho \leq \frac{dC}{dt} \leq 1 + \rho$$

se puede decir que el cronómetro está funcionando dentro de su especificación. La constante ρ es especificada por el fabricante, y se conoce como **velocidad máxima de flujo**. Observe que la velocidad máxima de flujo especifica hasta qué punto se permiten fluctuaciones en la distorsión del reloj. En la figura 6-5 se ilustran relojes lentos, perfectos, y rápidos.

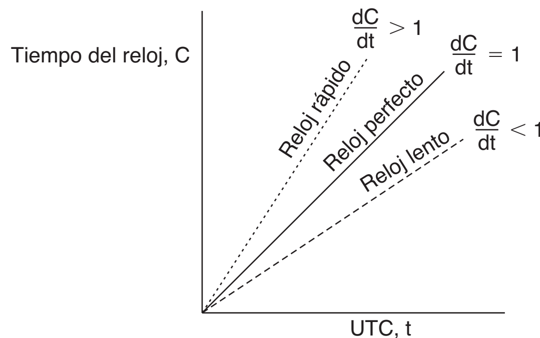


Figura 6-5. Relación entre el tiempo del reloj y el UTC cuando el reloj marca a diferentes velocidades.

Si dos relojes parten del UTC en la dirección opuesta, en un tiempo Dt después de ser sincronizados pueden estar, cuando mucho, a $2\rho\ \Delta t$ de diferencia. Si los diseñadores del sistema operativo quieren garantizar que dos relojes nunca difieran por más de δ, los relojes deben ser sincronizados nuevamente (en el software) al menos cada $\delta/2\rho$ segundos. Los diversos algoritmos difieren precisamente en cómo se lleva a cabo esta resincronización.

Protocolo de tiempo de red

Un método común en muchos protocolos, originalmente propuesto por Cristian (1989), es dejar a los clientes contactar a un servidor de tiempo. El último puede proporcionar exactamente el tiempo actual, por ejemplo, debido a que está equipado con un receptor WWV o un reloj exacto. Por supuesto, el problema cuando se contacta al servidor es que los retrasos del mensaje ocasionarán que el tiempo reportado no esté actualizado. El truco reside en encontrar una buena estimación para estos retrasos. Considere la situación esquematizada en la figura 6-6.

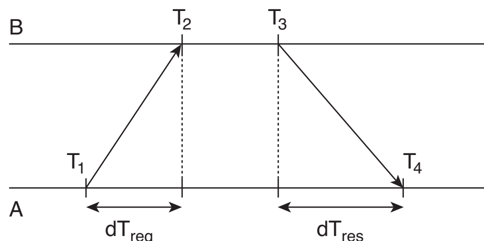


Figura 6-6. Obtención del tiempo actual desde un servidor de tiempo.

En este caso, A enviará una petición a B, con un registro de tiempo T_1. B, a su vez, registrará el tiempo del receptor, T_2 (tomado de su propio reloj local), y devolverá una respuesta con un registro de tiempo T_3, y encimará el valor T_2 previamente registrado. Por último, A registra el tiempo de la llegada de la respuesta, T_4. Supongamos que el retraso de la propagación de A a B es casi el mismo que de B a A, ello significa que $T_2 - T_1 \approx T_4 - T_3$. En ese caso, A puede estimar su compensación con respecto a B como

$$\theta = T_3 - \frac{(T_2 - T_1) + (T_4 - T_3)}{2} = \frac{(T_2 - T_1) + (T_3 - T_4)}{2}$$

Por supuesto, al tiempo no se le permite ir hacia atrás. Si un reloj A es rápido, $\theta < 0$, significa que A debe, en principio, configurar su reloj hacia atrás. Esto no está permitido ya que podría causar problemas serios, tales como que un archivo objeto compilado justamente después de un cambio en el reloj tuviera un tiempo anterior al de la fuente, el cual se modificó justamente antes del cambio en el reloj.

Tal cambio debe introducirse gradualmente. Una forma de hacerlo es como sigue. Supongamos que el cronómetro se configura para generar 100 interrupciones por segundo. Normalmente, cada interrupción añadiría 10 ms al tiempo. Cuando disminuye la velocidad, la rutina de interrupción añade sólo 9 ms cada vez hasta que se realiza la corrección. De manera similar, el reloj puede ser adelantado gradualmente añadiendo 11 ms en cada interrupción en lugar de adelantarlo de una sola vez.

En el caso del **protocolo de tiempo de red** (**NTP**, por sus siglas en inglés), éste se configura en pares entre servidores. En otras palabras, B también sondeará a A en cuanto a su tiempo actual. La compensación θ se calcula tal como ya indicamos, junto con la estimación δ para el retraso:

$$\delta = \frac{(T_2 - T_1) + (T_4 - T_3)}{2}$$

Ocho pares de valores (θ, δ) se almacenan en el bufer, para tomar finalmente el valor mínimo encontrado para δ como la mejor estimación del retraso entre los dos servidores, y subsecuentemente el valor asociado θ como la estimación más confiable de la compensación.

Aplicar el NTP simétricamente debe, en principio, permitir también a *B* ajustar su reloj con el de *A*. Sin embargo, si se sabe que el reloj de *B* es más exacto, entonces tal ajuste sería imprudente. Para resolver este problema, el NTP divide a los servidores en estratos. Un servidor con un **reloj de referencia**, tal como un receptor WWV o un reloj atómico, se conoce como **servidor de estrato 1** (se dice que el propio reloj opera en el estrato 0). Cuando *A* contacte a *B*, sólo ajustará su tiempo si su propio estrato es más alto que el de *B*. Además, después de la sincronización, el estrato de *A* será un nivel más alto que el de *B*. En otras palabras, si *B* es un servidor de estrato *k*, entonces *A* se volverá un servidor de estrato ($k + 1$) si el nivel de su estrato original ya era mayor que *k*. Debido a la simetría del NTP, si el nivel del estrato de *A* era *menor* que el de *B*, entonces *B* se autoajustará con *A*.

Existen muchas características importantes del NTP, de las cuales una gran cantidad se relaciona con identificación y enmascaramiento de errores, pero también con ataques a la seguridad. Mills (1992) describe al NTP, y es reconocido por lograr (a nivel mundial) una exactitud en el rango de 1 a 50 ms. La versión más reciente (NTPv4) fue documentada inicialmente sólo mediante su implementación, pero ahora puede encontrarse una descripción detallada en Mills (2006).

Algoritmo de Berkeley

En muchos algoritmos como el de NTP, el servidor de tiempo es pasivo. Otras máquinas solicitan periódicamente el tiempo. Todo lo que se hace es responder a sus consultas. En Berkeley UNIX se aplica exactamente el método opuesto (Gusella y Zatti, 1989). Aquí, el servidor de tiempo (de hecho, un demonio de tiempo) es activo, ya que cada cierto tiempo pregunta a cada máquina sobre la hora ahí registrada. Basado en las respuestas, calcula un tiempo promedio y les indica a todas las máquinas que adelanten o atrasen sus relojes, según la nueva hora. Este método es conveniente para un sistema en el que ninguna máquina tiene un receptor WWV. El operador debe configurar manualmente y de manera periódica la hora del demonio de tiempo. La figura 6-7 ilustra el método.

En la figura 6-7(a), a las 3:00, el demonio de tiempo indica a las otras máquinas su hora y les pregunta la hora que tienen. En la figura 6-7(b), las máquinas responden qué tanto se encuentran adelante o detrás del demonio de tiempo. Armado con estos números, el demonio calcula el promedio e indica a cada máquina cómo ajustar su reloj [vea la figura 6-7(c)].

Observe que, para muchos propósitos, es suficiente con que todas las máquinas coincidan en la misma hora. No resulta esencial que esta hora también coincida con el tiempo real tal como se anuncia por radio cada cierto tiempo. Si en nuestro ejemplo de la figura 6-7 el reloj del demonio de tiempo nunca se hubiera calibrado manualmente, no se generaría daño alguno ya que ninguno de los otros

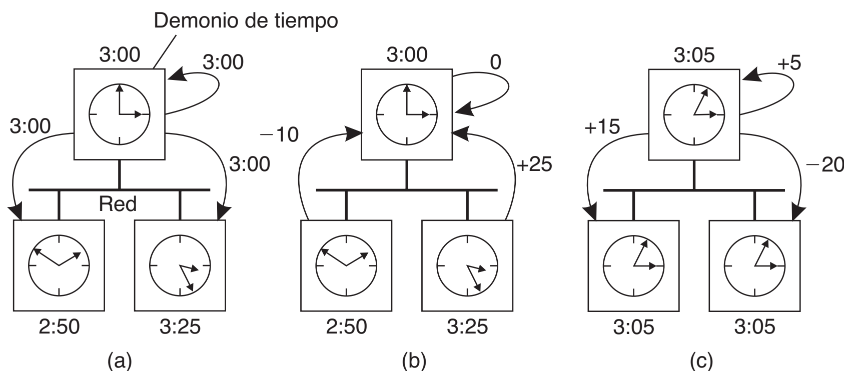


Figura 6-7. (a) El demonio de tiempo pregunta a las otras máquinas los valores de sus relojes. (b) Las máquinas responden. (c) El demonio de tiempo les indica cómo ajustar sus relojes.

nodos se comunica con computadoras externas. Todos coincidirán felizmente en una hora, sin que ese valor tenga relación alguna con la realidad.

Sincronización de relojes en redes inalámbricas

Una ventaja importante de los sistemas distribuidos más tradicionales es que podemos utilizar los servidores de tiempo de manera más sencilla y eficiente. Además, la mayoría de las máquinas pueden contactarse entre sí, lo que permite diseminar información de modo relativamente sencillo. Estas suposiciones ya no son válidas en muchas redes inalámbricas, principalmente en redes de monitoreo. Los nodos son recursos restringidos, y el enrutamiento multisaltos es caro. Además, con frecuencia es importante optimizar algoritmos para el consumo de energía. Éstas y otras observaciones han dado pie al diseño de muchos diferentes algoritmos de sincronización de relojes para redes inalámbricas. A continuación consideraremos una solución específica. Sivrikaya y Yener (2004) proporcionan una breve panorámica de otras soluciones. Puede encontrarse una amplia investigación en Sundararaman y colaboradores (2005).

La **sincronización de transmisión de referencias** (**RBS**, por sus siglas en inglés) es un protocolo de sincronización de relojes que difiere mucho de otras propuestas (Elson y cols., 2002). Primero, el protocolo no asume que hay un solo nodo con una cuenta exacta del tiempo real disponible. En lugar de ayudar a proporcionar a todos los nodos el tiempo UTC, este protocolo ayuda simplemente a sincronizar internamente los relojes, justo como el algoritmo de Berkeley. Segundo, las soluciones explicadas hasta el momento están diseñadas para lograr que el emisor y el receptor estén sincronizados, esencialmente siguiendo un protocolo de dos vías. El RBS se desvía de este patrón haciendo que sólo los receptores se sincronicen, y manteniendo al emisor fuera del ciclo.

En el RBS, un emisor transmite un mensaje de referencia que permitirá a sus receptores ajustar sus relojes. Una observación clave es que en una red de monitoreo el tiempo para propagar una

señal hacia otros nodos es más o menos constante, debido a que no se asume ningún enrutamiento multisaltos. En este caso el tiempo de propagación se mide desde el momento en que el mensaje abandona la interfaz de la red del emisor. En consecuencia, dos importantes fuentes de variación en la transferencia del mensaje dejan de jugar un papel importante en la estimación de retrasos: el tiempo dedicado a construir el mensaje, y el tiempo dedicado a acceder a la red. Este principio aparece en la figura 6-8.

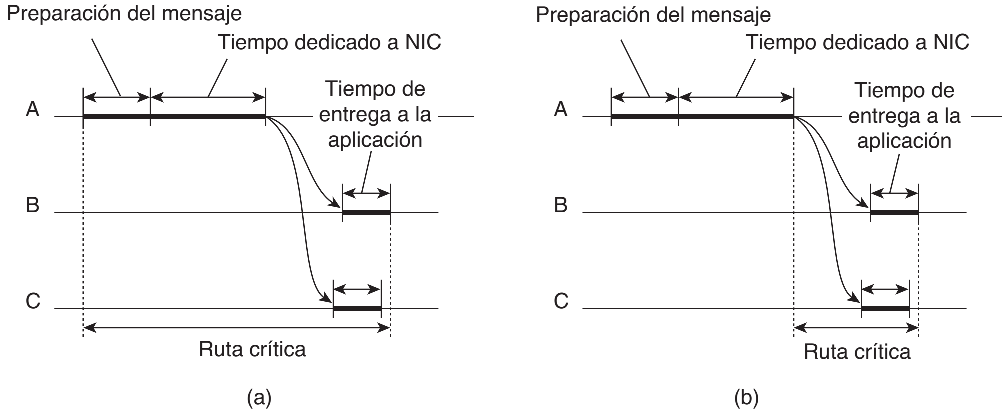


Figura 6-8. (a) Ruta crítica usual para determinar los retrasos de la red. (b) Ruta crítica en el caso de RBS.

Observe que en protocolos como el NTP, se agrega un registro de tiempo al mensaje antes de que se pase a la interfaz de la red. Más aún, como las redes inalámbricas se basan en un protocolo de contención, en general no se sabe cuánto tiempo llevará poder transmitir un mensaje. Estos factores no deterministas se eliminan en el RBS. Lo que queda es el tiempo de entrega al receptor, pero este tiempo varía considerablemente en menor medida que el tiempo de acceso a la red.

La idea subyacente al RBS es simple: cuando un nodo transmite un mensaje de referencia *m*, cada nodo *p* simplemente registra el tiempo $T_{p,m}$ recibido por *m*. Observe que $T_{p,m}$ se lee desde el reloj local de *p*. Al ignorar el desajuste del reloj, dos nodos *p* y *q* pueden intercambiar entre sí sus tiempos de entrega para estimar sus propias compensaciones relativas:

$$Compensación\,[\pi, \theta] = \frac{\sum_{k=1}^{M}(T_{p,k} - T_{q,k})}{M}$$

donde *M* es el número total de mensajes de referencia enviados. Esta información es importante: el nodo *p* sabrá el valor del reloj de *q* con respecto a su propio valor. Más aún, si dicho nodo simplemente almacena estas compensaciones, no hay necesidad de ajustar su propio reloj, lo cual ahorra energía.

Por desgracia, los relojes pueden avanzar de manera diferente. El efecto es que un simple cálculo de la compensación promedio, como hicimos antes, no funcionará: los últimos valores

enviados son sólo menos exactos que los primeros. Además, conforme el tiempo pasa, se supone que la compensación se incrementa. Elson y colaboradores utilizan un algoritmo muy simple para arreglar esto: en lugar de calcular un promedio, aplican la regresión lineal estándar para calcular la compensación como una función:

$$\textit{Compensación}\ [p, q](t) = \alpha t + \beta$$

Las constantes α y β se calculan a partir de los pares ($T_{p,\,k}$, $T_{q,\,k}$). Esta nueva forma permitirá un cálculo mucho más preciso del valor actual del reloj de *q* por parte del nodo *p*, y viceversa.

6.2 RELOJES LÓGICOS

Hasta el momento hemos supuesto que la sincronización de relojes está naturalmente relacionada con el tiempo real. Sin embargo, también hemos visto que puede ser suficiente que cada nodo coincida con un tiempo actual, sin que el tiempo sea necesariamente el mismo que el tiempo real. Podemos ir un paso más adelante. Por ejemplo, para ejecutar *make*, resulta adecuado que dos nodos acuerden que *input.o* sea actualizado por una nueva versión de *input.c*. En este caso, dar seguimiento a los otros eventos de cada uno (como producir una nueva versión de *input.c*) es lo que importa. Para estos algoritmos, es una convención referirse a los relojes como **relojes lógicos**.

En un artículo clásico, Lamport (1978) mostró que aunque la sincronización de relojes es posible, no necesita ser absoluta. Si dos procesos no interactúan, no es necesario que sus relojes sean sincronizados ya que la falta de sincronización no se notaría y, por tanto, no ocasionaría problemas. Más aún, señaló que lo generalmente importante no es que todos los procesos coincidan exactamente en el tiempo, sino que coincidan en el orden en que ocurren los eventos. En el ejemplo de *make*, lo que cuenta es si *input.c* es más antiguo o más reciente que *input.o*, y no sus tiempos absolutos de creación.

En esta sección explicaremos el algoritmo de Lamport, el cual sincroniza los relojes lógicos. También explicaremos una extensión del método de Lamport, la cual se conoce como registro de tiempo vectorial.

6.2.1 Relojes lógicos de Lamport

Para sincronizar los relojes lógicos, Lamport definió una relación llamada **ocurrencia anterior**. La expresión $a \rightarrow b$ se lee como "*a* ocurre antes que *b*", y significa que todos los procesos coinciden en que ocurre el primer evento *a* y, después de eso, ocurre el evento *b*. La relación ocurrencia anterior puede observarse directamente en dos situaciones:

1. Si *a* y *b* son eventos del mismo proceso, y *a* ocurre antes que *b*, entonces $a \rightarrow b$ es verdadera.

2. Si *a* es el evento en el que un proceso envía un mensaje, y *b* es el evento de recepción del mensaje por otro proceso, entonces $a \rightarrow b$ también es verdadero. Un mensaje no

puede recibirse antes de ser enviado, o incluso al mismo tiempo en que es enviado, ya que necesita cierta cantidad de tiempo finita, diferente de cero, para llegar.

La ocurrencia anterior es una relación transitiva, por lo que si $a \to b$ y $b \to c$, entonces $a \to c$. Si dos eventos, x y y, ocurren en diferentes procesos que no intercambian mensajes (ni siquiera indirectamente a través de terceras partes), entonces $x \to y$ no es verdadera, pero tampoco $y \to x$. Se dice que estos eventos son **concurrentes**, lo cual simplemente significa que nada se puede decir (o necesita decir) sobre cuándo ocurrieron estos eventos, o sobre cuál evento ocurrió primero.

Lo que necesitamos es una manera de medir la noción de tiempo en tal forma que a cada evento, a, podamos asignarle un valor de tiempo $C(a)$ en el que todos los procesos coincidan. Estos valores de tiempo deben tener la propiedad de que si $a \to b$, entonces $C(a) < C(b)$. Para reformular las condiciones que establecimos antes, si a y b son dos eventos dentro del mismo proceso, y a ocurre antes que b, entonces $C(a) < C(b)$. De manera similar, si a es el envío de un mensaje realizado por un proceso y b es la recepción de ese mensaje por otro proceso, entonces $C(a)$ y $C(b)$ deben asignarse de tal manera que todos coincidan con los valores de $C(a)$ y $C(b)$, con $C(a) < C(b)$. Además, el tiempo del reloj, C, siempre debe ir hacia delante (incrementándose), nunca hacia atrás (disminuyendo). Es posible hacer correcciones al tiempo agregando un valor positivo, nunca quitando uno.

Ahora veamos el algoritmo de Lamport propuesto para asignar tiempos a eventos. Consideremos los tres procesos delineados en la figura 6-9(a). Los procesos se ejecutan en diferentes máquinas, cada una con su propio reloj, avanzando a su propia velocidad. Como podemos ver en la figura, cuando el reloj ha hecho marca 6 veces en el proceso P_1, éste ha hecho marca 8 veces en el proceso P_2, y 10 veces en el proceso P_3. Cada reloj avanza a velocidad constante, pero las velocidades son diferentes debido a las diferencias en los cristales.

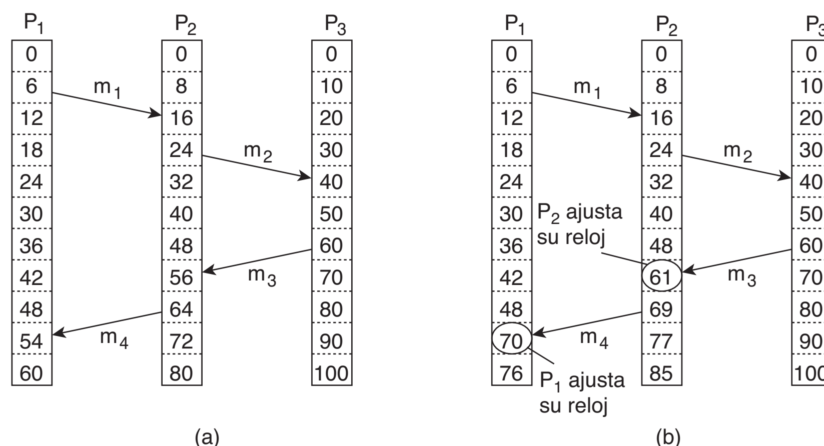


Figura 6-9. (a) Tres procesos, cada uno con su propio reloj. Los relojes avanzan a velocidades diferentes. (b) El algoritmo de Lamport corrige los relojes.

Al tiempo 6, el proceso P_1 envía el mensaje m_1 al proceso P_2. Cuánto tiempo demore en llegar el mensaje depende del reloj al que se le crea. En cualquier evento, el reloj del proceso P_2 dice 16 cuando llega el mensaje. Si el mensaje lleva inscrito el tiempo de inicio 6, el proceso P_2 concluirá que le tomó 10 marcas realizar el recorrido. Este valor es, desde luego, posible. De acuerdo con este razonamiento, el mensaje m_2 desde P_2 hasta R se lleva 16 marcas, de nuevo un valor plausible.

Ahora consideremos el mensaje m_3. Éste deja el proceso P_3 al 60 y llega a P_2 al 56. De manera similar, el mensaje m_4 desde P_2 hasta P_1 sale en el 64 y llega en el 54. Estos valores son claramente imposibles. Es tal situación la que debe evitarse.

La solución de Lamport se deriva directamente a partir de la relación ocurrencia-anterior. Debido a que m_3 salió en el 60, debe llegar en el 61 o después. Por tanto, cada mensaje lleva el tiempo de envío de acuerdo con el reloj del remitente. Cuando un mensaje llega y el reloj del destinatario muestra un valor anterior al tiempo en que el mensaje fue enviado, el destinatario rápidamente adelanta su reloj para estar una unidad adelante del tiempo de envío. En la figura 6-9(b) observamos que m_3 ahora llega en el 61. De manera similar, m_4 llega en el 70.

Con el propósito de prepararnos para abordar la explicación sobre relojes vectoriales, formulemos este procedimiento de manera más precisa. En este punto, es importante diferenciar tres capas de software distintas, como ya vimos en el capítulo 1: la red, la capa middleware, y una capa de aplicación, según muestra la figura 6-10. Lo siguiente es típico de la capa middleware.

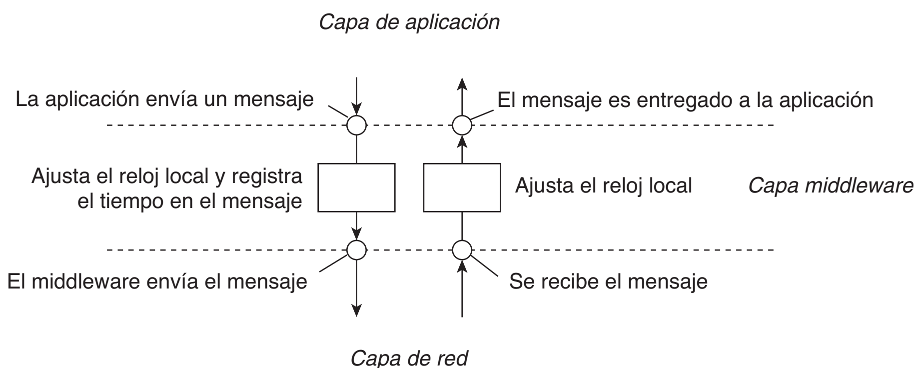


Figura 6-10. Posicionamiento de los relojes lógicos de Lamport en sistemas distribuidos.

Para implementar los relojes lógicos de Lamport, cada proceso P_i mantiene un contador local C_i. Estos contadores se actualizan de acuerdo con los siguientes pasos (Raynal y Singhal, 1996):

1. Antes de ejecutar un evento (es decir, enviar un mensaje a través de la red, entregar un mensaje a una aplicación, o algún otro evento interno), P_i ejecuta $C_i \leftarrow C_i + 1$.

2. Cuando el proceso P_i envía un mensaje m a P_j, éste ajusta el registro de tiempo de m, $ts(m)$, igual a C_i después de haber ejecutado el paso anterior.

3. Una vez que se recibe el mensaje m, el proceso P_j ajusta su propio contador local como $C_j \leftarrow \text{máx}\{C_j, ts(m)\}$, después de lo cual ejecuta el primer paso y entrega el mensaje a la aplicación.

En algunas situaciones, es deseable un requerimiento adicional: dos eventos nunca deben ocurrir exactamente al mismo tiempo. Para lograr este objetivo podemos incluir el número del proceso en el que ocurre el evento al final del tiempo de menor orden, separado por un punto decimal. Por ejemplo, un evento en el tiempo 40 del proceso P_i será registrado como $40.i$.

Observe que al asignar el tiempo del evento $C(a) \leftarrow C_i(a)$ si a ocurrió en el proceso P_i al tiempo $C_i(a)$, tenemos una implementación distribuida del valor de tiempo global que originalmente buscábamos.

Ejemplo: transmisión totalmente ordenada

Como una aplicación de relojes lógicos de Lamport, consideremos la situación en que una base de datos se ha replicado a través de varios sitios. Por ejemplo, para mejorar el rendimiento de las consultas, un banco puede colocar copias de una base de datos contable en dos ciudades diferentes, digamos Nueva York y San Francisco. Una consulta siempre se reenvía a la copia más cercana. El precio de una respuesta rápida a una consulta es el de altos costos de actualización, debido a que cada operación de actualización debe realizarse en cada réplica.

De hecho, hay un requerimiento más estricto con respecto a las actualizaciones. Supongamos que un cliente basado en San Francisco desea agregar \$100 a su cuenta, la cual actualmente contiene \$1 000. Al mismo tiempo, en Nueva York un empleado del banco inicia una actualización mediante la cual la cuenta del cliente se verá incrementada con el 1% de interés. Ambas actualizaciones deben realizarse en ambas copias de la base de datos. Sin embargo, debido a los retrasos en la comunicación de la red subyacente, las actualizaciones pueden llegar en el orden que muestra la figura 6-11.

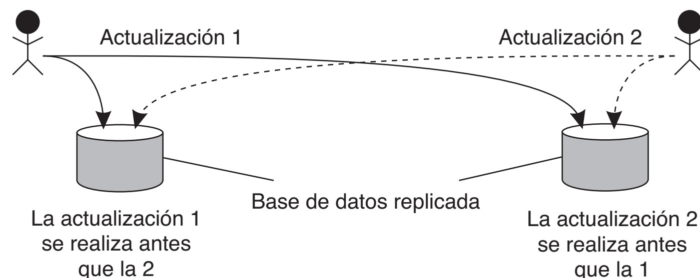


Figura 6-11. Actualización de una base de datos replicada, dejándola en un estado inconsistente.

La operación de actualización del cliente se realiza en San Francisco antes de la actualización del interés. Por contraste, la copia de la cuenta basada en Nueva York se actualiza primero con el

1% de interés, y después con el depósito de $100. En consecuencia, la base de datos de San Francisco registrará un monto total de $1 111, mientras que la base de datos de Nueva York registra $1 110.

El problema que enfrentamos es que las dos operaciones de actualización debieron haberse realizado en el mismo orden en cada copia. Aunque existe una diferencia entre que el depósito se procese antes de la actualización del interés o viceversa, el orden que se sigue no es importante desde el punto de vista de consistencia. La cuestión importante es que ambas copias deben ser exactamente las mismas. En general, situaciones como éstas requieren una **transmisión totalmente ordenada**, es decir, una operación de transmisión mediante la cual todos los mensajes se entreguen en el mismo orden a cada destinatario. Los relojes lógicos de Lamport pueden utilizarse para implementar transmisiones totalmente ordenadas de modo completamente distribuido.

Consideremos un grupo de procesos que transmiten mensajes entre sí. Cada mensaje se registra siempre con el tiempo actual (lógico) de su remitente. Cuando se transmite un mensaje, también es conceptualmente enviado al remitente. Además, suponemos que los mensajes del mismo remitente se reciben en el orden en que fueron enviados, y que ningún mensaje se pierde.

Cuando un proceso recibe un mensaje, éste se coloca en una cola local, y se ordena de acuerdo con su registro de tiempo. El destinatario transmite un acuse de recibo a los otros procesos. Observe que si seguimos el algoritmo de Lamport para ajustar los relojes locales, el registro de tiempo del mensaje recibido es menor que el registro de tiempo del acuse. El aspecto interesante de este método es que todos los procesos tienen, en algún momento, la misma copia de la cola local (debido a que ningún mensaje es eliminado).

Un proceso puede entregar un mensaje de la cola a la aplicación que está ejecutando sólo cuando ese mensaje se encuentra a la cabeza de la cola, y si todos los demás procesos recibieron un acuse. En ese punto, el mensaje es eliminado de la cola y entregado a la aplicación; los acuses asociados pueden simplemente eliminarse. Debido a que cada proceso tiene la misma copia de la cola, todos los mensajes se entregan en el mismo orden en todas partes. En otras palabras, hemos establecido una transmisión totalmente ordenada.

Como veremos en capítulos posteriores, la transmisión totalmente ordenada es un medio importante para servicios replicados en los cuales las réplicas se mantienen consistentes dejándolas ejecutar las mismas operaciones en el mismo orden en todas partes. Como las réplicas básicamente siguen las mismas transiciones en el mismo estado finito de la máquina, también se conocen como **replicación de estado de máquina** (Schneider, 1990).

6.2.2 Relojes vectoriales

Los relojes lógicos de Lamport dieron pie a una situación en la que todos los eventos de un sistema distribuido se ordenan completamente según la propiedad de que, si el evento *a* ocurrió antes que el evento *b*, entonces *a* también estará posicionado en un orden anterior a *b*, es decir, $C(a) < C(b)$.

Sin embargo, con los relojes de Lamport, nada puede decirse sobre la relación entre dos eventos *a* y *b* por mera comparación de sus valores de tiempo $C(a)$ y $C(b)$, respectivamente. En otras

palabras, si $C(a) < C(b)$, entonces esto no necesariamente implica que a realmente ocurrió antes que b; se necesita algo más para afirmar eso.

Para explicarlo, consideremos los mensajes enviados por los tres procesos que muestra la figura 6-12. Denotamos como $T_{env}(m_i)$ al tiempo lógico en que se envió el mensaje m_i, y de igual modo, como $T_{rec}(m_i)$ al tiempo de recepción. Por construcción, sabemos que para cada mensaje $T_{env}(m_i) < T_{rec}(m_i)$. ¿Pero qué podemos concluir en general a partir de $T_{env}(m_i) < T_{rec}(m_j)$?

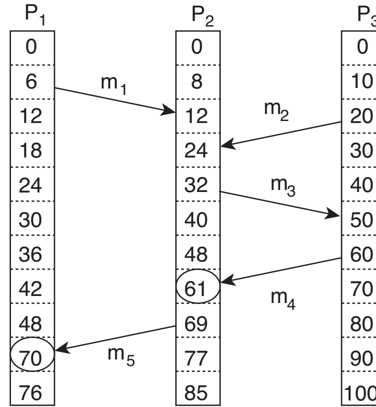


Figura 6-12. Transmisión concurrente de mensajes utilizando relojes lógicos.

En el caso para el cual $m_i = m_1$ y $m_j = m_3$, sabemos que estos valores corresponden a los eventos ocurridos en el proceso P_2, ello significa que m_3 en realidad fue enviado después de la recepción del mensaje m_1. Esto puede indicar que el envío del mensaje m_3 dependió de lo recibido con el mensaje m_1. Sin embargo, también sabemos que $T_{rec}(m_1) < T_{env}(m_2)$. Pero el envío de m_2 no tiene que ver con la recepción de m_1.

El problema es que los relojes de Lamport no capturan la **causalidad**. La causalidad puede capturarse mediante los **relojes vectoriales**. Un reloj vectorial, $VC(a)$, asignado a un evento a, tiene la propiedad de que si $VC(a) < VC(b)$ para algún evento b, entonces se sabe que el evento a precede en causalidad al evento b. Los relojes vectoriales se construyen de manera que cada proceso P_i mantenga un vector VC_i con las dos siguientes propiedades:

1. $VC_i[i]$ es el número de eventos que han ocurrido hasta el momento en P_i. En otras palabras, $VC_i[i]$ es el reloj lógico del proceso P_i.

2. Si $VC_i[j] = k$, entonces P_i sabe que han ocurrido k eventos en P_j. Así, éste es el conocimiento de P_i del tiempo local en P_j.

La primera propiedad se mantiene incrementando $VC_i[i]$ ante la ocurrencia de cada nuevo evento en el proceso P_i. La segunda propiedad se mantiene encimando los vectores junto con los mensajes que se envían. En particular, se realizan los siguientes pasos:

1. Antes de ejecutar un evento (es decir, enviar un mensaje por la red, entregar un mensaje a una aplicación, o algún otro evento interno), P_i ejecuta $VC_i[i] \leftarrow VC_i[i] + 1$.

2. Cuando el proceso P_i envía un mensaje m a P_j, éste establece el registro de tiempo de m, $ts(m)$, igual a VC_i después de haber ejecutado el paso anterior.

3. Una vez que se recibe el mensaje m, el proceso P_j ajusta su propio vector configurando $VC_j[k] \leftarrow \text{máx}\{VC_j[k], ts(m)[k]\}$ para cada k, después de lo cual ejecuta el primer paso y libera el mensaje a la aplicación.

Observemos que si un evento a tiene un registro de tiempo $ts(a)$, entonces $ts(a)[i] - 1$ denota el número de eventos procesados en P_i que preceden en causalidad a a. En consecuencia, cuando P_j recibe un mensaje de P_i con un registro de tiempo $ts(m)$, éste sabe el número de eventos que han ocurrido en P_i y que preceden en causalidad el envío de m. Sin embargo, es más importante que a P_j se le informe también sobre cuántos eventos han ocurrido en *otros* procesos antes de que P_i envíe el mensaje m. En otras palabras, el registro de tiempo $ts(m)$ le indica al destinatario cuántos eventos de otros procesos han precedido al envío de m, y de cuáles m puede depender causalmente.

Imposición de la comunicación causal

Al utilizar relojes vectoriales, ahora ya es posible garantizar que un mensaje sea entregado sólo si todos los mensajes que causalmente lo preceden también han sido recibidos. Para lograr tal esquema, supondremos que los mensajes se transmiten dentro de un grupo de procesos. Observemos que esta **transmisión causalmente ordenada** es más débil que la transmisión totalmente ordenada que explicamos antes. En específico, si dos mensajes no están relacionados de ninguna manera, no nos interesa el orden en que se entregan a las aplicaciones; pueden incluso entregarse en diferente orden en diferentes ubicaciones.

Además, suponemos que los relojes sólo se ajustan cuando se envían y reciben mensajes. En particular, una vez que envía un mensaje, el proceso P_i sólo incrementará en 1 a $VC_i[i]$. Cuando éste recibe un mensaje m con registro de tiempo $ts(m)$, éste sólo ajustará $VC_i[k]$ a $\text{máx}\{VC_j[k], ts(m)[k]\}$ para cada k.

Ahora supongamos que P_j recibe un mensaje m desde P_i con un registro de tiempo (vectorial) $ts(m)$. La entrega del mensaje a la capa de aplicación será entonces retrasada hasta que las dos siguientes condiciones se satisfagan:

1. $ts(m)[i] = VC_j[i] + 1$

2. $ts(m)[k] \leq VC_j[k]$ para toda $k \neq i$

La primera condición establece que *m* es el siguiente mensaje que P_j esperaba del proceso P_i. La segunda condición establece que P_j ha visto todos los mensajes que ha visto P_i cuando éste envió el mensaje *m*. Observe que no hay necesidad de que el proceso P_j retrase la entrega de sus propios mensajes.

Como un ejemplo, consideremos tres procesos P_0, P_1, y P_2, como se muestra en la figura 6-13. En el tiempo local (1,0,0), P_1 envía un mensaje *m* a los otros dos procesos. Después de que lo recibe P_1, este último decide enviar *m**, el cual llega a P_2 más rápido que *m*. En ese punto, la entrega de *m** es retrasada por P_2 hasta que *m* haya sido recibida y entregada a la capa de aplicación de P_2.

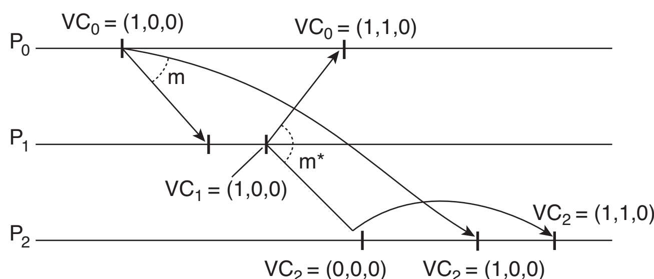


Figura 6-13. Imposición de la comunicación causal.

Una nota sobre la entrega ordenada de mensajes

Algunos sistemas middleware, principalmente ISIS y su sucesor Horus (Birman y Van Renesse, 1994), proporcionan soporte para la transmisión totalmente ordenada y causalmente ordenada (confiable). Ha habido cierta controversia sobre si tal soporte debe proporcionarse como parte de la capa de comunicación de mensajes, o si las aplicaciones deben manejar el ordenamiento (vea, por ejemplo, Cheriton y Skeen, 1993; y Birman, 1994). Las cosas no se han resuelto, pero lo más importante es que los argumentos aún son válidos.

Existen dos problemas principales con dejar que el middleware trate con el ordenamiento de mensajes. Primero, debido a que el middleware no puede decir qué contiene un mensaje, sólo captura la *potencial* causalidad. Por ejemplo, dos mensajes del mismo remitente, totalmente independientes, siempre serán marcados según la causalidad relacionada por la capa middleware. Este método es excesivamente restrictivo y puede ocasionar problemas de eficiencia.

Un segundo problema es que no toda la causalidad puede captarse. Consideremos un tablero electrónico de comunicación. Supongamos que Alicia publica un artículo. Si ella telefonea a Bob y le informa sobre lo que acaba de escribir, Bob puede publicar otro artículo en respuesta sin haber visto la publicación de Alicia en el tablero. En otras palabras, hay causalidad entre la publicación de Bob y la de Alicia debido a la comunicación *externa*. El sistema del tablero de comunicación no capta esta causalidad.

En esencia, las cuestiones de ordenamiento, como muchas otras cuestiones de comunicación específica de la aplicación, pueden resolverse adecuadamente examinando la aplicación en la que se lleva a cabo dicha comunicación. A esto se le conoce también como **argumento fin a fin** en los sistemas de diseño (Saltzer y cols., 1984). Una desventaja de tener únicamente soluciones al nivel de aplicación es que el desarrollador se ve obligado a concentrarse en las cuestiones que no se relacionan de inmediato con la funcionalidad central de la aplicación. Por ejemplo, el ordenamiento puede no ser el problema más importante cuando se desarrolla un sistema de mensajes tal como un tablero electrónico de comunicación. En ese caso, tener una capa de comunicación subyacente que maneje el ordenamiento puede resultar conveniente. Abordaremos el argumento fin a fin varias veces, principalmente cuando tratemos con la seguridad en los sistemas distribuidos.

6.3 EXCLUSIÓN MUTUA

Para los sistemas distribuidos resultan fundamentales la concurrencia y la colaboración entre diversos procesos. En muchos casos, esto significa también que los procesos necesitarán de acceso simultáneo a los mismos recursos. Para evitar que tales accesos concurrentes corrompan los recursos, o que los vuelvan inconsistentes, se necesita encontrar soluciones que garanticen que los procesos tengan acceso mutuamente exclusivo. En esta sección veremos algunos de los algoritmos distribuidos más importantes que se han propuesto. Saxena y Rai (2003) proporcionan una investigación reciente sobre algoritmos distribuidos para exclusión mutua; anterior, pero aún importante, es el trabajo de Velázquez (1993).

6.3.1 Visión general

Los algoritmos distribuidos de exclusión mutua pueden clasificarse en dos diferentes categorías. En las **soluciones basadas en token**, la exclusión mutua se logra pasando entre los procesos un mensaje especial conocido como **token**. Sólo hay un token disponible, y quien lo tenga puede acceder al recurso compartido. Cuando termina, el token pasa al siguiente proceso. Si un proceso tiene el token pero no está interesado en acceder al recurso, simplemente lo pasa.

Las soluciones basadas en token tienen algunas propiedades importantes. Primero, de acuerdo con la organización de los procesos, éstos pueden garantizar fácilmente que todos tendrán la oportunidad de acceder a los recursos. En otras palabras, evitan la **inanición**. Segundo, el **interbloqueo** mediante los cuales diversos procesos se esperan unos a otros para continuar pueden evitarse fácilmente, contribuyendo a su simplicidad. Por desgracia, el principal inconveniente de las soluciones basadas en token es que, cuando el token se pierde (por ejemplo, debido a que falla el proceso que lo tiene), es necesario iniciar un intrincado proceso distribuido para garantizar la creación de un nuevo token, pero sobre todo, para que sea el único token.

Como alternativa, muchos algoritmos distribuidos de exclusión mutua se derivan de un **método basado en permisos**. En este caso, un proceso que desea el primer acceso a los recursos requiere

el permiso de los otros procesos. Hay muchas formas de garantizar tal permiso, y en las siguientes secciones consideraremos algunas.

6.3.2 Un algoritmo centralizado

En un sistema distribuido, la manera más directa de lograr la exclusión mutua es simular lo que se hace en un sistema de un procesador. Se elige un proceso como coordinador. Siempre que un proceso desea acceder a un recurso compartido, envía un mensaje de petición al coordinador mencionando el recurso al que desea acceder, y solicita permiso. Si ningún otro proceso está accediendo al recurso en ese momento, el coordinador devuelve una respuesta en la que otorga el permiso, como se muestra en la figura 6-14(a). Cuando la respuesta llega, el proceso solicitante puede continuar.

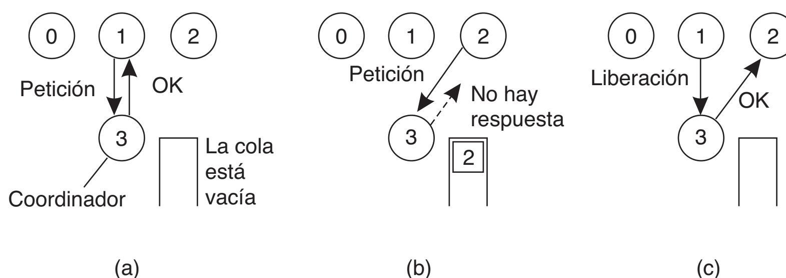


Figura 6-14. El proceso 1 solicita permiso al coordinador para acceder a un recurso compartido. El permiso es otorgado. (b) Luego el proceso 2 solicita permiso para acceder al mismo recurso. El coordinador no responde. (c) Cuando el proceso 1 libera el recurso, éste se lo indica al coordinador, que después le responde a 2.

Ahora supongamos que otro proceso, 2 en la figura 6-14(b), solicita permiso para acceder al recurso. El coordinador sabe que otro proceso ya se encuentra con ese recurso, por lo que no puede otorgar el permiso. El método exacto utilizado para negar el permiso es dependiente del sistema. En la figura 6-14(b), el coordinador simplemente se abstiene de responder y bloquea al proceso 2, que está en espera de una respuesta. Como alternativa, pudo enviarse una respuesta que dijera "permiso negado". De cualquier manera, por el momento el coordinador pone en cola la petición de 2 y espera más mensajes.

Cuando el proceso 1 termina con el recurso, envía un mensaje al coordinador para que libere su acceso exclusivo, como ilustra la figura 6-14(c). El coordinador toma el primer elemento de la cola de peticiones aplazadas y envía a ese proceso un mensaje de autorización. Si el proceso estuviera aún bloqueado (es decir, éste es el primer mensaje para él), se desbloquea y accede al recurso. Si ya se ha enviado un mensaje explícito, en el que se niega el permiso, el proceso tiene que sondear el tráfico entrante o bloquearse después. De cualquier manera, cuando ve la autorización, puede continuar.

Es fácil advertir que el algoritmo garantiza la exclusión mutua: el coordinador sólo permite a un proceso a la vez acceder al recurso. También es justo, ya que las peticiones son autorizadas en el orden en que se reciben. Ningún proceso espera por siempre (no hay inanición). Es fácil implementar el esquema, y sólo requiere tres mensajes para utilizar un recurso (petición, autorización, liberación). Su simplicidad lo convierte en una solución atractiva para muchas situaciones prácticas.

El método centralizado también tiene defectos. El coordinador es un solo punto de falla, por lo que si falla, todo el sistema puede irse abajo. Si los procesos normalmente se bloquean después de hacer una petición, no pueden distinguir un coordinador desactivado de un "permiso negado" ya que, en ambos casos, ningún mensaje regresa. Además, en un sistema grande, un solo coordinador puede volverse un cuello de botella en cuanto a rendimiento. Sin embargo, los beneficios provenientes de su simplicidad pesan más que las potenciales desventajas. Asimismo, las soluciones distribuidas no necesariamente son mejores, como veremos en los siguientes ejemplos.

6.3.3 Un algoritmo descentralizado

Con frecuencia, tener un solo coordinador es un mal método. Demos un vistazo a una solución totalmente descentralizada. Lin y colaboradores (2004) proponen el uso de un algoritmo de votación que puede ejecutarse con un sistema basado en DHT. En esencia, su solución se extiende al coordinador central de la siguiente manera. Se supone que cada recurso se replica n veces. Cada réplica tiene su propio coordinador para controlar el acceso de procesos concurrentes.

Sin embargo, siempre que un proceso desee acceder al recurso, éste simplemente tendrá que lograr una votación mayoritaria a partir de $m > n/2$ coordinadores. A diferencia del esquema centralizado que explicamos antes, asumimos que cuando un coordinador no otorga el permiso para acceder a un recurso (lo que hará cuando haya otorgado el permiso a otro proceso), se lo informa al solicitante.

Este esquema básicamente hace que la solución original centralizada sea menos vulnerable ante las fallas de un solo coordinador. La suposición es que cuando un coordinador falla, se recupera rápidamente pero habrá olvidado cualquier voto otorgado antes de fallar. Otra forma de ver esto es que el coordinador se reinicia a sí mismo en cualquier momento. El riesgo que corremos es que un reinicio hará que el coordinador olvide los permisos otorgados previamente a algunos procesos para acceder al recurso. En consecuencia, después de su recuperación puede nuevamente otorgar, de manera incorrecta, permiso a otro proceso.

Sea p la probabilidad de que un coordinador se reinicie durante un intervalo de tiempo Δt. La probabilidad, $P[k]$, de que cada k de m coordinadores se reinicie durante el mismo intervalo es

$$P[k] = \begin{bmatrix} m \\ k \end{bmatrix} p^k \, (1 - p)^{m-k}$$

Dado que al menos $2m - n$ coordinadores necesitan reiniciarse para violar la corrección del mecanismo de votación, la probabilidad de que tal violación ocurra es entonces $\sum_{k=2m-n}^{n} P[k]$. Para

ilustrar lo que esto significaría, suponga que estamos tratando con un sistema basado en DHT donde cada nodo participa alrededor de 3 horas en una fila. Sea Δt igual a 10 segundos, lo cual se considera un valor conservador para un solo proceso que quiere acceder a un recurso compartido. (Se necesitan diferentes mecanismos para tareas muy largas.) Con $n = 32$ y $m = 0.75\ n$, la probabilidad de violar la corrección es menor que 10 240. Esta probabilidad es, con certeza, menor que la disponibilidad de cualquier recurso.

Para implementar este esquema, Lin y colaboradores (2004) utilizan un sistema basado en DHT en el que un recurso se replica n veces. Supongamos que el recurso se conoce con su nombre único, *rnombre*. Podemos entonces suponer que a la i-ésima réplica se le llama *rnombre-i*, la cual se utiliza después para calcular una clave única mediante una función hash conocida. En consecuencia, todo proceso puede generar las n claves dado el nombre de un recurso, y posteriormente buscar a cada nodo responsable de una réplica (y controlar el acceso total a esa réplica).

Si se niega el permiso para acceder al recurso (es decir, un proceso obtiene menos de m votos), se supone que desistirá durante cierto tiempo elegido al azar, y lo intentará más tarde. El problema con este esquema es que si muchos nodos desean acceder al mismo recurso, desde luego que su uso decae rápidamente. En otras palabras, hay tantos nodos compitiendo por ganar el acceso que en algún momento ninguno podrá obtener suficientes votos, y dejarán sin utilizar al recurso. En Lin y colaboradores (2004) podemos encontrar una solución para resolver este problema.

6.3.4 Un algoritmo distribuido

Para muchos, contar con un algoritmo probabilísticamente correcto simplemente no es suficiente. De modo que los investigadores han elaborado algoritmos distribuidos deterministas de exclusión mutua. En 1978, el artículo de Lamport sobre la sincronización de relojes presentó el primero de tales algoritmos. Ricart y Agrawala (1981) lo hicieron más eficiente. En esta sección describiremos su método.

El algoritmo de Ricart y Agrawala requiere un ordenamiento total de todos los eventos del sistema. Es decir, para cualquier par de eventos, tales como los mensajes, debe ser inequívoco cuál de ellos realmente ocurre primero. El algoritmo de Lamport, presentado en la sección 6.2.1, es una manera de lograr este orden y puede utilizarse para proporcionar registros de tiempo para la exclusión mutua distribuida.

El algoritmo funciona de la siguiente manera. Cuando un proceso desea acceder a un recurso compartido, elabora un mensaje que contiene el nombre del recurso, su número de proceso, y el tiempo actual (lógico). Entonces envía el mensaje a todos los demás procesos, incluyéndose de manera conceptual. Se supone que el envío de los mensajes es confiable; es decir, no se pierde mensaje alguno.

Cuando un proceso recibe un mensaje de petición de otro proceso, la acción que tome dependerá de su propio estado con respecto al recurso mencionado en el mensaje. Se deben distinguir claramente tres casos:

1. Si el destinatario no accede al recurso y no desea acceder a él, envía un mensaje de *OK* al remitente.

2. Si el destinatario ya cuenta con acceso al recurso, simplemente no responde. En vez de eso, coloca la petición en una cola.

3. Si el receptor también quiere acceder al recurso, pero aún no lo ha hecho, compara el registro de tiempo del mensaje entrante con el del mensaje que ya ha enviado a todos. El menor gana. Si el mensaje entrante tiene un registro de tiempo menor, el receptor envía de vuelta un mensaje de *OK*. Si su propio mensaje tiene un registro menor, el destinatario coloca en la cola el mensaje entrante y no envía nada.

Después de pedir permiso y enviar las peticiones, un proceso espera hasta que todos los demás tengan permiso. Tan pronto como todos los permisos están dentro, el proceso puede continuar. Cuando termina, envía un mensaje de *OK* a todos los procesos de su cola y elimina todos los demás.

Intentemos comprender por qué funciona el algoritmo. Si no existe conflicto alguno, funciona de manera clara. Sin embargo, supongamos que dos procesos intentan acceder al recurso de manera simultánea, como podemos ver en la figura 6-15(a).

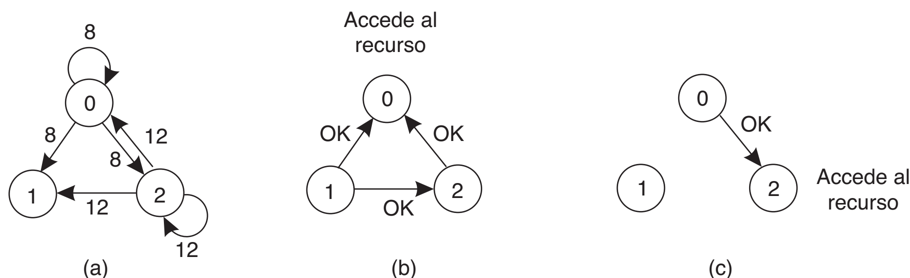


Figura 6-15. (a) Dos procesos desean acceder a un recurso compartido en el mismo momento. (b) El proceso 0 contiene el menor registro de tiempo, de modo que gana. (c) Cuando el proceso 0 se lleva a cabo, envía también un *OK*, de modo que el 2 puede seguir adelante.

El proceso 0 envía a todos una petición con un registro de tiempo de 8, mientras que al mismo tiempo, el proceso 2 envía a todos una petición con registro de 12. El proceso 1 no está interesado en el recurso, de manera que envía *OK* a ambos remitentes. Los procesos 0 y 2 ven el conflicto y comparan los registros de tiempo. El proceso 2 ve que perdió, de manera que le concede permiso a 0 al enviar *OK*. Ahora el proceso 0 forma en la cola a la petición de 2, para su posterior procesamiento y acceso al recurso, según muestra la figura 6-15(b). Cuando termina, elimina la petición de 2 de su cola y envía un mensaje de *OK* para procesar 2, permitiendo a este último seguir adelante,

como ilustra la figura 6-15(c). El algoritmo funciona debido a que, en caso de un conflicto, el menor registro de tiempo gana y todos los demás coinciden en el orden de los registros.

Observe que la situación ilustrada en la figura 6-15 habría sido diferente si el proceso 2 hubiera enviado su mensaje antes, de modo que el proceso 0 lo hubiera obtenido y cedido el permiso antes de hacer su propia petición. En este caso, 2 habría notado que él mismo ya tenía acceso al recurso al momento de la petición, y la hubiera formado en la cola en lugar de enviar una respuesta.

Como en el algoritmo centralizado que explicamos antes, la exclusión mutua se garantiza sin interbloqueos o inanición. El número de mensajes requerido por entrada ahora es $2(n - 1)$, en donde n es el número total de procesos. Lo mejor de todo es que no existe un punto de falla.

Por desgracia, el único punto de falla se reemplazó por n puntos de falla. Si alguno de los procesos falla, fallará en responder las peticiones. Este silencio se interpretará (de manera incorrecta) como una negativa de permiso, lo cual bloqueará todos los intentos posteriores de los procesos para entrar a las regiones críticas. Dado que la probabilidad de fallo de uno de los n procesos es al menos n veces tan grande como una simple falla de coordinación, nos las hemos arreglado para reemplazar un algoritmo poco eficaz con uno que es más de n veces peor y requiere también mucho más tráfico de red.

El algoritmo se puede parchar mediante el mismo truco que propusimos antes. Cuando llega una petición, el destinatario siempre envía una respuesta, ya sea autorizando o negando el permiso. Cada vez que se pierde una petición o una respuesta, el remitente agota el tiempo y continúa hasta que obtiene una respuesta, o el remitente concluye que el destinatario está desactivado. Después de que se niega una petición, el remitente debe lanzar un bloqueo y esperar un mensaje posterior de *OK*.

Otro problema con este algoritmo es que se debe utilizar ya sea una primitiva de comunicación por multitransmisión, o que cada proceso deba mantener una lista de membresía de grupo por sí solo, incluyendo a los procesos que entran al grupo, que abandonan el grupo, y que fallan. El método funciona mejor con grupos pequeños de procesos que nunca modifican sus membresías de grupo.

Por último, recuerde que uno de los problemas con el algoritmo centralizado es que hacerse cargo de todas las peticiones puede provocar un cuello de botella. En el algoritmo distribuido, *todos* los procesos están involucrados en *todas* las decisiones relacionadas con el acceso a un recurso compartido. Si un proceso es incapaz de manipular la carga, es poco probable que forzar a todos los procesos a hacer exactamente lo mismo en paralelo ayude mucho.

Son posibles muchas mejoras menores para este algoritmo. Por ejemplo, obtener permiso de todos es realmente exagerado. Todo lo que se requiere es un método para prevenir que dos procesos accedan al recurso al mismo tiempo. El algoritmo puede modificarse para obtener el permiso cuando se han juntado la mayoría de los permisos de los demás procesos, y no de todos ellos. Por supuesto, en esta variación, después de que un proceso ha logrado el permiso para un proceso, no puede obtener el mismo permiso para otro proceso sino hasta que termine el primero.

No obstante, este algoritmo es más lento, más complicado, más caro, y menos robusto que el original centralizado. ¿Por qué molestarse en estudiarlo bajo estas condiciones? Por una razón, muestra que al menos un algoritmo distribuido es posible; algo que no resultaba evidente cuando

comenzamos. Además, al indicar los atajos, estimulamos a los futuros teóricos a intentar reproducir los algoritmos que realmente son útiles. Finalmente, así como comer espinacas y aprender latín en la escuela preparatoria, se dice que, de manera abstracta, algunas cosas son buenas para usted. Podría tomar algún tiempo descubrir exactamente cuáles.

6.3.5 Un algoritmo de anillo de token

Un método completamente distinto para lograr de manera determinística la exclusión mutua aparece en la figura 6-16. Aquí tenemos una red de bus, como podemos ver en la figura 6-16(a) (por ejemplo, ethernet), sin orden inherente a alguno de los procesos. En software, un anillo lógico se construye con cada proceso asignado a una posición en el anillo, como lo muestra la figura 6-16(b). En el anillo, las posiciones se pueden localizar con el orden numérico de las direcciones de red o por otros medios. No importa cuál es el orden. Todo lo que importa es que cada proceso sabe cuál es el siguiente después de él.

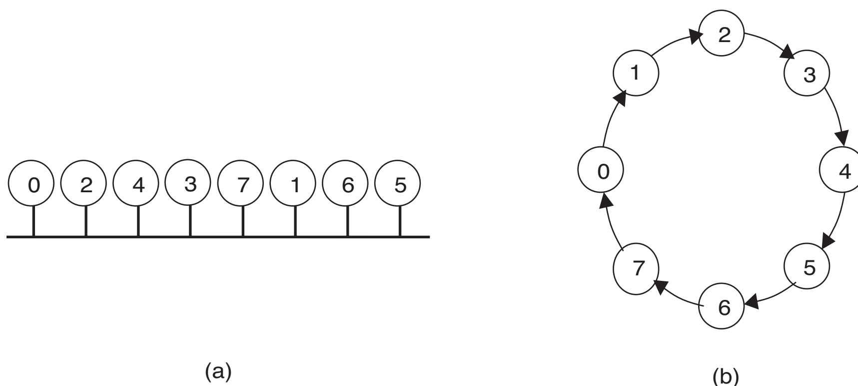


Figura 6-16. (a) Un grupo desordenado de procesos en una red. (b) Un anillo lógico construido en el software.

Cuando se inicia el anillo, al proceso 0 se le asigna un **token**. El token circula alrededor del anillo. Se pasa desde el proceso k al proceso $k + 1$ (modula el tamaño del anillo) en los mensajes punto a punto. Cuando un proceso adquiere el token de su vecino, verifica si necesita acceder al recurso compartido. Si es así, el proceso sigue adelante, hace el trabajo que requiere hacer, y libera los recursos. Una vez que ha terminado, pasa el token a lo largo del anillo. No está permitido acceder de inmediato de nuevo al recurso mediante el uso del mismo token.

Si un proceso manipula el token de su vecino y no está interesado en el recurso, solamente lo pasa a través del anillo. En consecuencia, cuando ningún proceso necesita el recurso, la señal sólo circula a alta velocidad por todo el anillo.

El grado de exactitud de este algoritmo es fácil de advertir. Solamente un proceso tiene el token en cualquier instante, de modo que sólo un proceso puede realmente llegar al recurso. Dado que

el token circula entre los procesos en un orden bien definido, la inanición no ocurrirá. Cuando un proceso decide que quiere acceder al recurso, lo peor que puede pasarle es que deba esperar a que todos los demás procesos utilicen el recurso.

Como es usual, este algoritmo también tiene problemas. Si el token se pierde en algún momento, deberá reponerse. De hecho, es difícil detectar su pérdida porque la cantidad de tiempo entre las apariciones sucesivas del token en la red no tiene límites. El hecho de que el token no haya sido visto durante una hora no significa que se ha perdido; alguien podría seguir utilizándolo.

El algoritmo también entra en problemas si falla un proceso, pero la recuperación es más fácil que en otros casos. Si requerimos que un proceso que recibe el token acuse de recibido, un proceso inactivo será detectado cuando su vecino intente darle el token y falle. En ese punto, el proceso inactivo se puede eliminar del grupo, y quien mantiene el token puede arrojarlo por encima del proceso inactivo hacia el miembro de la línea contiguo, o al siguiente, si es necesario. Por supuesto, hacer eso requiere que todos mantengan la configuración actual del anillo.

6.3.6 Comparación de los cuatro algoritmos

Una breve comparación de los cuatro algoritmos de exclusión mutua que hemos visto sería realmente instructiva. En la figura 6-17 listamos los algoritmos y tres propiedades clave: el número de mensajes requeridos para que un proceso acceda y libere un recurso compartido, el retraso antes de que pueda darse el acceso (si suponemos que los mensajes se pasan de manera secuencial sobre una red), y algunos problemas asociados con cada algoritmo.

Algoritmo	Mensajes por entrada/salida	Retraso antes de la entrada (durante el tiempo del mensaje)	Problemas
Centralizado	3	2	Falla el coordinador
Descentralizado	3mk, k = 1, 2, ...	2 m	Innanición, baja eficiencia
Distribuido	2(n − 1)	2(n − 1)	Falla de cualquier proceso
Anillo de token	1 a ∞	0 a n − 1	Pérdida del token, falla del proceso

Figura 6-17. Comparación de tres algoritmos de exclusión mutua.

El algoritmo centralizado es el más simple y también el más eficiente. Sólo requiere de tres mensajes para entrar y salir de una región crítica: una petición, un permiso para entrar, y una liberación para salir. En el caso descentralizado, vemos que estos mensajes necesitan ser realizados por cada uno de los *m* coordinadores, pero ahora es posible que se necesiten varios intentos (para ello introducimos la variable *k*). El algoritmo distribuido requiere $n - 1$ mensajes de petición, uno para cada uno de los demás procesos, y los $n - 1$ mensajes adicionales de autorización, para un total de $2(n - 1)$. (Suponemos que solamente se utilizan los canales de comunicación punto a punto.) Con

el algoritmo de anillo de token, el número es variable. Si cada proceso requiere entrar de manera constante a una región crítica, entonces cada token arrojará como resultado una entrada y una salida, para un promedio de un mensaje por región crítica introducida. Por otra parte, en ocasiones el token puede circular por horas sin que ningún proceso se interese en él. En este caso, el número de mensajes por entrada a una región crítica es ilimitado.

El retraso desde el momento en que un proceso requiere entrar a una región crítica hasta su entrada real también varía para los tres algoritmos. Cuando el tiempo para utilizar un recurso es corto, el factor dominante en el retraso es el mecanismo real para acceder a un recurso. Cuando los recursos se utilizan por un periodo largo, el factor dominante es la espera para que todos los demás tomen su turno. En la figura 6-17 mostramos el caso anterior. En el caso centralizado, toma solamente dos mensajes entrar a una región crítica, pero toma $3mk$ veces para el caso descentralizado, en donde k es el número de intentos que se requiere hacer. Si asumimos que los mensajes se envían uno después del otro, se requieren $2(n - 1)$ mensajes para el caso distribuido. Para el anillo de token, el tiempo varía desde 0 (el token acaba de llegar) hasta $n - 1$ (el token acaba de partir).

Por último, todos los algoritmos excepto el descentralizado sufren mucho en el caso de una falla. Debemos incluir medidas especiales y complejidad adicional para evitar que una falla tire todo el sistema. Es irónico que los algoritmos distribuidos sean incluso más sensibles a las fallas que el centralizado. En un sistema diseñado para ser tolerante a fallas, ninguno de los algoritmos anteriores sería adecuado, pero si las fallas son muy frecuentes pudieran ajustarse. El algoritmo descentralizado es menos sensible a fallas, pero los procesos pudieran sufrir de inanición y se requieren medidas especiales para garantizar su eficiencia.

6.4 POSICIONAMIENTO GLOBAL DE LOS NODOS

Cuando aumenta el número de nodos en un sistema distribuido, se vuelve cada vez más difícil para cualquier nodo dar seguimiento a los demás. Dicho conocimiento puede ser importante para la ejecución de algoritmos distribuidos tales como el enrutamiento, la multitransmisión, la colocación de datos, la búsqueda, etc. Ya vimos diferentes ejemplos en los que se organizan grandes colecciones de nodos en topologías específicas que facilitan la ejecución eficiente de dichos algoritmos. En esta sección, daremos un vistazo a otra organización relacionada con asuntos de tiempo.

En **redes geométricas sobrepuestas**, a cada nodo se le asigna una posición dentro de un espacio geométrico m-dimensional, tal que la distancia entre dos nodos en dicho espacio refleja una métrica de rendimiento en el mundo real. El ejemplo más simple, y más aplicado, es en donde la distancia se corresponde con la latencia internodal. En otras palabras, dados dos nodos P y O, entonces la distancia $d(P,Q)$ refleja el tiempo que le toma a un mensaje viajar desde P hacia Q y *viceversa*.

Existen muchas aplicaciones para las redes geométricas. Considere una situación en donde un sitio web en el servidor O es replicado en múltiples servidores $S_1,\ldots,S_k$ en internet. Cuando un cliente C solicita una página desde O, este último pudiera decidir redireccionar una petición

hacia el servidor más cercano a *C*, esto es, aquel que da el mejor tiempo de respuesta. Si conocemos la ubicación geométrica de *C*, así como la de cada réplica del servidor, entonces *O* puede simplemente elegir al servidor S_i para el cual $d(C,S_i)$ es mínima. Observemos que dicha selección solamente requiere un procesamiento local en *O*. En otras palabras, no existe, por ejemplo, necesidad de mostrar todas las latencias entre *C* y cada uno de los servidores replicados.

Otro ejemplo, el cual trabajaremos con detalle en el siguiente capítulo, es la ubicación óptima de la réplica. Considere de nuevo un sitio web que ha ganado cierta posición entre sus clientes. Si el sitio fuera a replicar su contenido hacia *K* servidores, puede calcular las *K* mejores posiciones en donde colocar las réplicas, de tal manera que el tiempo de respuesta promedio de cliente por réplica sea mínimo. Realizar dichos cálculos es casi trivialmente posible si clientes y servidores tienen posiciones geométricas que reflejan las latencias internodales.

Como último ejemplo, considere el **enrutamiento basado en posición** (Araujo y Rodrigues, 2005; y Stojmenovic, 2002). En dichos esquemas, se reenvía un mensaje hacia su destino solamente para posicionar la información. Por ejemplo, un inocente algoritmo de enrutamiento para permitir que cada nodo reenvíe un mensaje hacia el vecino más cercano a su destino. Aunque podemos mostrar fácilmente que un algoritmo específico no necesita converger, si ilustra que solamente se utiliza la información para tomar una decisión. No hay necesidad de propagar la información de enlace o para todos los nodos de la red, como es el caso con los algoritmos tradicionales de enrutamiento.

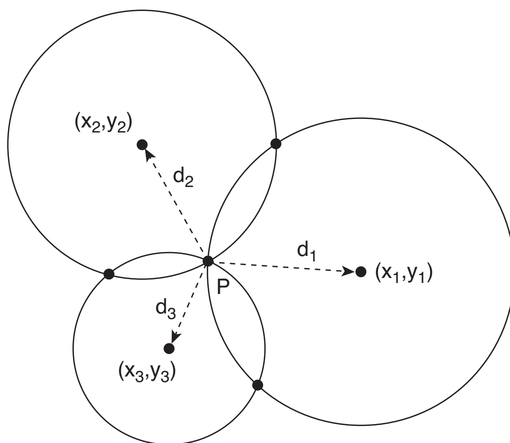


Figura 6-18. Cálculo de la posición de los nodos en un espacio bidimensional.

En teoría, posicionar un nodo en un espacio geométrico *m*-dimensional requiere medidas de distancia $m + 1$ para los nodos con posiciones conocidas. Esto puede verse fácilmente al considerar el caso $m = 2$, como ilustra la figura 6-18. Si asumimos que el nodo *P* quiere calcular su propia posición, contacta otros tres nodos con posiciones conocidas y mide su distancia hacia cada uno

de ellos. Hacer contacto solamente con un nodo le dirá a *P* el círculo sobre el que se localiza; hacer contacto con sólo dos nodos le informará con respecto a la posición de la intersección de los dos círculos (lo que por lo general consta de dos puntos); contactar un tercer nodo permitirá de manera subsiguiente que *P* calcule su ubicación real.

Tal como en el GPS, el nodo *P* puede calcular sus propias coordenadas (x_P, y_P) mediante la resolución de las tres ecuaciones con las incógnitas x_P y y_P:

$$d_i = \sqrt{(x_i - x_P)^2 + (y_i - y_P)^2} \qquad (i = 1, 2, 3)$$

Como se ha dicho, por lo general d_i corresponde a la medida de latencia entre *P* y el nodo en (x_i, y_i). Esta latencia se puede estimar como la mitad del retraso de ciclo, pero debiera quedar claro que su valor será diferente con el tiempo. El efecto es un posicionamiento diferente cada vez que *P* quiera recalcular su posición. Más aún, si otros nodos fueran a utilizar la posición actual de *P* para calcular sus propias coordenadas, entonces debería quedar claro que el error en la posición de *P* afectará la certeza de la posición de los demás nodos.

Más aún, debería quedar claro también que las distancias medidas por diferentes nodos, por lo general, no son consistentes. Por ejemplo, asumimos que calculamos distancias en un espacio de una dimensión tal como aparece en la figura 6-19. En este ejemplo, vemos que aunque *R* mide su distancia a *Q* como 2.0, y la medida de $d(P,Q)$ es 1.0, cuando *R* mide a $d(P,R)$ resulta ser 3.2, lo cual claramente es inconsistente con las otras dos medidas.

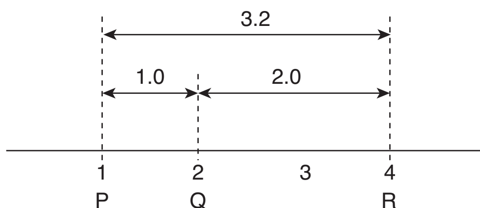


Figura 6-19. Medidas de distancia inconsistentes en un espacio unidimensional.

La figura 6-19 sugiere también la manera en que podemos mejorar esta situación. En nuestro ejemplo, podríamos resolver las inconsistencias simplemente calculando las posiciones en un espacio bidimensional. Sin embargo, esto por sí solo no resulta ser una solución general cuando se trata con muchas mediciones. De hecho, considerando las medidas de latencia en internet pudiera violar la **desigualdad del triángulo**, por lo general es imposible resolver completamente las inconsistencias. La desigualdad del triángulo establece que en un espacio geométrico, para cualquier tercia de nodos cualesquiera *P*, *Q* y *R*, siempre debe ser verdadero que $d(P,R) \leq d(P,Q) + d(Q,R)$.

Existen diversos métodos para enfrentar estos problemas. Un método, propuesto por Ng y Zhang (2002) es el uso de los nodos especiales de *L* $b_1, \ldots, b_L$, conocidos como **puntos de referencia**. Los puntos de referencia miden las latencias entre pares $d(b_i,b_j)$ y subsecuentemente

permiten que el nodo central calcule las coordenadas de cada punto de referencia. Hasta este punto, el nodo central busca minimizar la siguiente función de error agregada:

$$\sum_{i=1}^{L} \sum_{j=i+1}^{L} \left[\frac{d(b_i, b_j) - \hat{d}(b_i, b_j)}{d(b_i, b_j)} \right]^2$$

donde $\hat{d}(b_i,b_j)$ corresponde a la *distancia geométrica*, esto es, la distancia entre los nodos b_i y b_j está posicionada.

El parámetro oculto al minimizar la función de error agregada es la dimensión m. Desde luego, tenemos que $L > m$, pero nada nos previene de elegir un valor para m que sea mucho más pequeño que L. En ese caso, un nodo P mide su distancia a cada uno de los puntos de referencia L y calcula sus coordenadas al minimizar

$$\sum_{i=1}^{L} \left[\frac{d(b_i, P) - \hat{d}(b_i, P)}{d(b_i, P)} \right]^2$$

Como es evidente, mediante puntos de referencia bien establecidos, m puede ser tan pequeño como 6 o 7, donde $\hat{d}(P,Q)$ no es más que un factor 2 diferente de su latencia $d(P,Q)$ real para los nodos arbitrarios P y Q (Szyamniak y cols., 2004).

Otra manera de enfrentar este problema es ver una colección de nodos como un sistema grande en el que los nodos se adjuntan entre sí a través de resortes. En este caso, $| d(P,Q) - \hat{d}(P,Q) |$ indica la distancia de desplazamiento de los nodos P y Q de manera relativa a la situación en la cual el sistema de resortes estaría en estado estable. Al permitir que cada nodo cambie (de manera ligera) su posición, es posible mostrar que el sistema convergirá de una u otra forma hacia una organización óptima en la cual el error agregado es mínimo. En Vivaldi se sigue este método, del cual podemos encontrar los detalles en Dabek y colaboradores (2004a).

6.5 ALGORITMOS DE ELECCIÓN

Muchos algoritmos distribuidos requieren que un proceso actúe como coordinador, iniciador, o que represente algún papel en especial. En general, no importa qué proceso tenga esta responsabilidad especial, pero alguno tiene que realizarla. En esta sección veremos algoritmos para elegir un coordinador (y utilizaremos éste como un nombre general para el proceso especial).

Si todos los procesos son exactamente iguales, sin características que los distingan, no hay manera de seleccionar a uno para que sea el especial. En consecuencia, supondremos que cada proceso tiene un número único, por ejemplo, su dirección de red (por simplicidad, supondremos un proceso por máquina). En general, la elección de algoritmos intenta localizar el proceso que tenga el número más grande y designarlo como coordinador. Los algoritmos difieren en la forma en que efectúan la localización.

Además, supondremos que cada proceso conoce el número de los otros procesos. Lo que los procesos no saben es cuáles están aumentando y cuáles disminuyendo. El objetivo de un algoritmo de elección es garantizar que cuando inicie una elección, ésta concluya con todos los procesos de acuerdo con el que será el nuevo coordinador. Hay muchos algoritmos y variaciones, de los cuales los más importantes se explican en los libros de texto de Lynch (1996) y Tel (2000), respectivamente.

6.5.1 Algoritmos de elección tradicional

Comencemos analizando dos algoritmos tradicionales de elección para tener una idea de lo que han estado haciendo grupos completos de investigadores en las décadas pasadas. En secciones posteriores veremos las nuevas aplicaciones para el problema de elección.

El algoritmo del abusón (Bully)

Como primer ejemplo, consideremos el **algoritmo del abusón**, concebido por García-Molina (1982). Cuando cualquier proceso advierte que el coordinador ya no está respondiendo peticiones, inicia una elección. Un proceso, *P*, celebra una elección de la siguiente manera:

1. *P* envía un mensaje de *ELECCIÓN* a todos los procesos con números superiores.
2. Si ningún proceso responde, *P* gana la elección y se convierte en el coordinador.
3. Si uno de los procesos superiores responde, toma el mando. El trabajo de *P* está hecho.

En cualquier momento, un proceso puede recibir un mensaje de *ELECCIÓN* de alguno de sus colegas con número menor. Cuando llega un mensaje de este tipo, el destinatario envía un mensaje de *OK* de vuelta al remitente para indicarle que está activo y que tomará el control. El destinatario celebra entonces una elección, a menos que ya tenga una. En algún momento, todos los procesos se rinden menos uno, que es entonces el nuevo coordinador. Éste anuncia su victoria enviando un mensaje a todos los procesos en el que les indica que a partir de ese momento es el nuevo coordinador.

Si un proceso que previamente había fallado se recupera, celebra una elección. Si sucede que el proceso con el número más grande está en ejecución, él ganará la elección y asumirá el trabajo del coordinador. Así, el muchacho más grande del pueblo siempre gana, de ahí el nombre de "algoritmo del abusón".

En la figura 6-20 vemos un ejemplo de cómo funciona el algoritmo del abusón. El grupo consiste en ocho procesos, numerados del 0 al 7. Anteriormente, el proceso 7 fue el coordinador, pero falló. El proceso 4 es el primero en advertirlo, por lo que envía un mensaje de *ELECCIÓN* a los demás procesos superiores, a saber 5, 6 y 7, como se muestra en la figura 6-20(a). Los procesos 5 y 6 responden con *OK*, según muestra la figura 6-20(b). Una vez obtenida la primera de estas respuestas, 4 sabe que su trabajo ha concluido. Sabe que uno de estos peces gordos se volverá coor-

dinador. Él simplemente se relaja y espera a ver cuál será el ganador (aunque en este punto puede hacer muy buenas suposiciones).

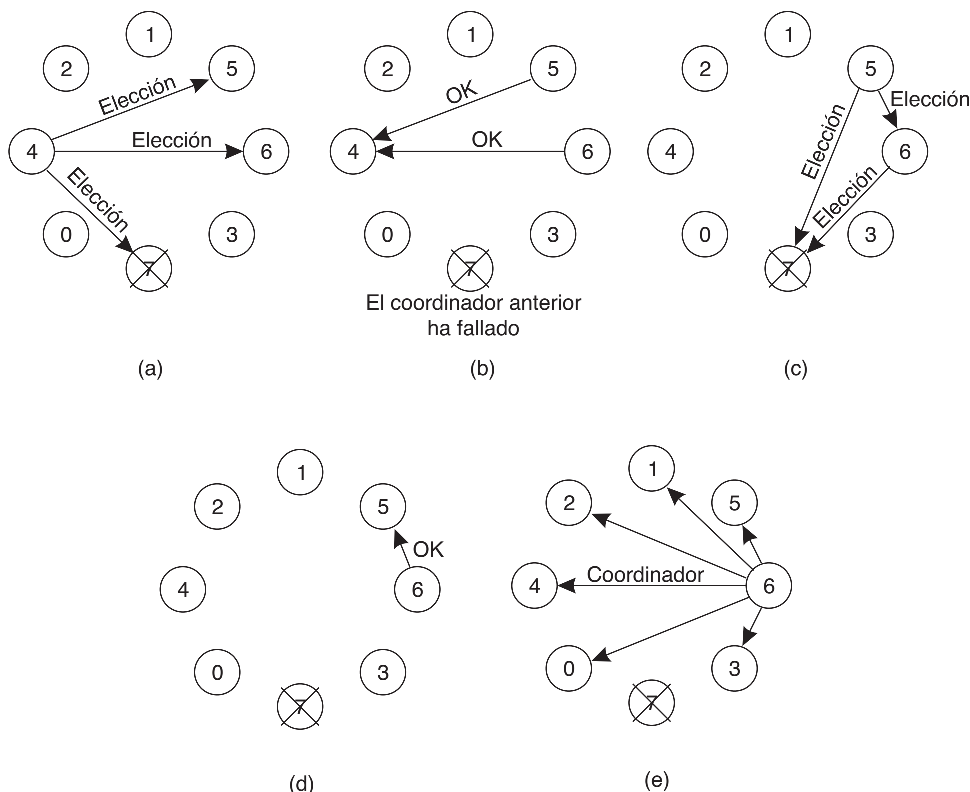


Figura 6-20. Algoritmo del abusón. (a) El proceso 4 celebra la elección. (b) Los procesos 5 y 6 responden, indicándole a 4 que se detenga. (c) Ahora 5 y 6 celebran una elección. (d) El proceso 6 le indica al 5 que se detenga. (e) El proceso 6 gana, y lo comunica a todos.

En la figura 6-20(c), tanto 5 como 6 celebran elecciones, cada uno envía mensajes sólo a los procesos superiores. En la figura 6-20(d), el proceso 6 le indica al 5 que se hará cargo. En este punto, 6 sabe que 7 está muerto y que él (6) es el ganador. Si es necesario recopilar información desde el disco o de cualquier otra parte, para continuar desde donde el coordinador anterior se quedó, ahora 6 debe hacer lo necesario. Cuando está listo para hacerse cargo, 6 lo anuncia enviando un mensaje de *COORDINADOR* a todos los procesos en ejecución. Cuando 4 recibe este mensaje, puede continuar con la operación que intentaba realizar cuando descubrió que 7 estaba inactivo, pero esta vez utilizando a 6 como coordinador. De esta manera se maneja la falla de 7, y el trabajo puede continuar.

Si el proceso 7 se reinicia en algún momento, simplemente enviará un mensaje de *COORDINADOR* a todos los demás y los obligará a someterse.

Un algoritmo de anillo

Otro algoritmo de elección se basa en el uso de un anillo. A diferencia de algunos algoritmos de anillo, éste no utiliza un token. Suponemos que los procesos están física o lógicamente ordenados, de tal forma que cada proceso sabe cuál es su sucesor. Cuando cualquier proceso advierte que el coordinador no funciona, elabora un mensaje de *ELECCIÓN* que contiene su propio número de proceso y envía el mensaje a su sucesor. Si el sucesor falló, el remitente lo salta y se dirige al siguiente miembro del anillo, al siguiente después de él, hasta que localice un proceso en ejecución. En cada paso del camino, el remitente agrega su propio número de proceso a la lista del mensaje para volverse un candidato a elegir como coordinador.

En algún momento, el mensaje regresa al proceso que inició todo. Ese proceso reconoce este evento cuando recibe un mensaje entrante que contiene su propio número de proceso. En ese punto, el tipo de mensaje cambia a *COORDINADOR* y circula una vez más, esta vez para informar a todos que es el coordinador (el miembro de la lista con el número mayor) y cuáles son los miembros del nuevo anillo. Cuando este mensaje ha circulado una vez, es eliminado y todos vuelven al trabajo.

Figura 6-21. Algoritmo de elección que utiliza un anillo.

En la figura 6-21 vemos lo que ocurre si dos procesos, 2 y 5, descubren simultáneamente que el coordinador anterior, el proceso 7, ha fallado. Cada proceso elabora un mensaje de *ELECCIÓN* y comienza a circular su mensaje, de modo independiente uno de otro. En algún momento, ambos mensajes recorrerán todo el camino, y tanto el proceso 2 como el 5 los convertirán en mensajes de *COORDINADOR* con exactamente los mismos miembros y en el mismo orden. Cuando ambos hayan hecho el recorrido nuevamente serán eliminados. Tener más mensajes en circulación no ocasiona daños; a lo sumo, consume un poco de ancho de banda, pero no se considera un desperdicio.

6.5.2 Elecciones en ambientes inalámbricos

Los algoritmos tradicionales de elección generalmente se basan en suposiciones que no son reales en ambientes inalámbricos. Por ejemplo, suponen que el paso de mensajes es confiable y que la topología de la red no cambia. Estas suposiciones son falsas en la mayoría de los ambientes inalámbricos, en especial en aquellos implementados para redes móviles a la medida.

Sólo se han desarrollado pocos protocolos para elecciones que funcionan en redes a la medida. Vasudevan y colaboradores (2004) proponen una solución que puede manejar nodos que fallan y redes particionadas. Una propiedad importante de su solución es que el *mejor* líder puede elegirse, en lugar de hacerlo al azar, como de alguna manera se hacía en las soluciones que explicamos anteriormente. Su protocolo funciona de la siguiente manera. Para simplificar nuestra explicación, sólo nos concentraremos en las redes a la medida, y pasaremos por alto que los nodos pueden moverse.

Considere una red inalámbrica a la medida. Para elegir un líder, cualquier nodo de la red, llamado fuente, puede iniciar una elección enviando un mensaje de *ELECCIÓN* a sus vecinos inmediatos (es decir, a los nodos de su rango). Cuando un nodo recibe una *ELECCIÓN* por primera vez, designa al remitente como su padre, y posteriormente envía un mensaje de *ELECCIÓN* a sus vecinos inmediatos, con excepción del padre. Cuando un nodo recibe un mensaje de *ELECCIÓN* de otro nodo que no sea su padre, simplemente acusa de recibido.

Cuando el nodo *R* ha designado al nodo *Q* como su padre, éste reenvía el mensaje de *ELECCIÓN* a sus vecinos inmediatos (excepto a *Q*) y espera la llegada de los acuses antes de enviar acuse de recibo del mensaje de *ELECCIÓN* de *Q*. Esta espera tiene una importante consecuencia. Primero, observe que los vecinos que ya han seleccionado un padre de inmediato responden a *R*. De manera más específica, si todos los vecinos ya tienen un padre, *R* es un nodo hoja y podrá responder rápidamente a *Q*. Al hacerlo, también reporta información tal como el tiempo de vida de la pila y otras capacidades de los recursos.

Esta información permitirá más tarde a *Q* comparar las capacidades de *R* con las de otros nodos, y seleccionar al mejor nodo para que sea el líder. Por supuesto, *Q* ha enviado un mensaje de *ELECCIÓN* sólo porque su padre, *P*, lo ha hecho también. A su vez, cuando *Q* en algún momento acuse la recepción del mensaje de *ELECCIÓN* previamente enviado por *P*, éste pasará también el nodo más elegible a *P*. De este modo, la fuente deberá saber en algún momento cuál es el mejor nodo para seleccionarlo como líder, y después transmitirá esta información a los demás nodos.

Este proceso aparece en la figura 6-22. Los nodos se han etiquetado de la *a* a la *k*, junto con su capacidad. El nodo *a* inicia una elección transmitiendo un mensaje de *ELECCIÓN* a los nodos *b* y *j*, como ilustra la figura 6-22(b). Después de ese paso, los mensajes de *ELECCIÓN* se propagan a todos los nodos, finalizando con la situación que aparece en la figura 6-22(e), donde hemos omitido las últimas transmisiones de los nodos *f* e *i*. A partir de ahí, cada nodo reporta a su padre el nodo con la mejor capacidad, como se muestra en la figura 6-22(f). Por ejemplo, cuando el nodo *g* recibe los acuses de sus hijos *e* y *h*, notará que *h* es el mejor nodo, y propagará [*h*, 8] a su propio padre, el nodo *b*. Al final, la fuente notará que *h* es el mejor líder y transmitirá esta información a los demás nodos.

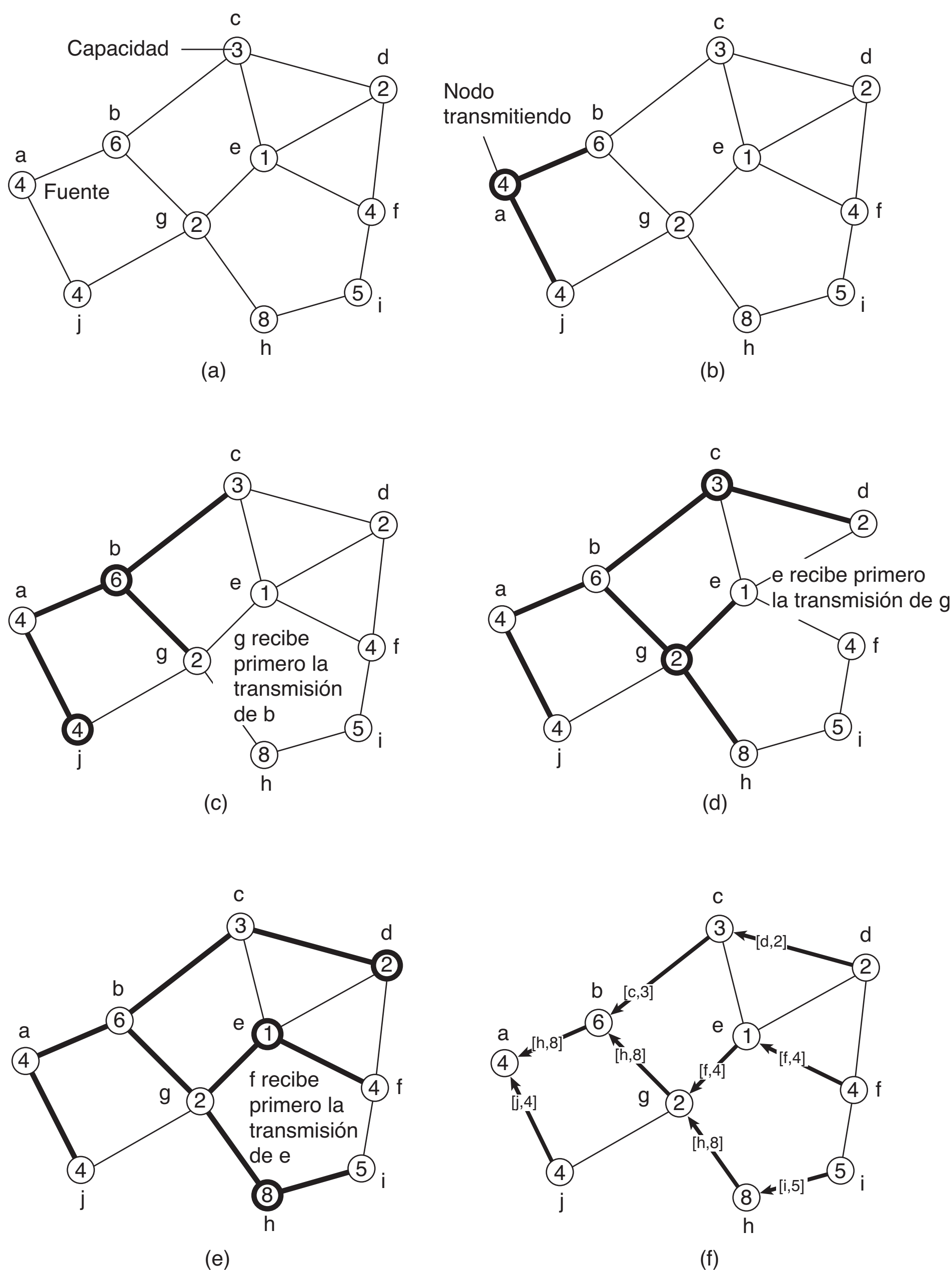


Figura 6-22. Algoritmo de elección en una red inalámbrica, con el nodo *a* como fuente. (a) Red inicial. (b) a (e) Fase de construcción de árbol (el último paso de transmisión de los nodos f e i no se muestra). (f) Se reporta a la fuente sobre el mejor nodo.

Cuando se inician varias elecciones, cada nodo decidirá unirse a una elección solamente. Con esta finalidad, cada fuente etiqueta su mensaje de *ELECCIÓN* con un identificador único. Los nodos participarán sólo en la elección con el identificador más alto, y detendrán cualquier participación actual en otras elecciones.

Con algunos ajustes menores, el protocolo puede ser mostrado para operar también cuando se particiona la red, y cuando los nodos se unen y se van. Podemos encontrar los detalles en Vasudevan y colaboradores (2004).

6.5.3 Elecciones en sistemas de gran escala

Los algoritmos que hemos explicado hasta ahora, generalmente, se aplican a sistemas distribuidos relativamente pequeños. Más aún, los algoritmos se concentran en la selección de un solo nodo. Hay situaciones en las que varios nodos deben seleccionarse, como en el caso de los **superpuntos** de las redes punto a punto, que explicamos en el capítulo 2. En esta sección nos concentramos específicamente en el problema de seleccionar superpuntos.

Lo y colaboradores (2005) identificaron los siguientes requerimientos que deben cumplirse para lograr la selección de superpuntos:

1. Los nodos normales deben tener acceso de baja latencia a los superpuntos.
2. Los superpuntos deben distribuirse uniformemente a través de la red sobrepuesta.
3. Debe haber una porción predefinida de superpuntos, relativa al número total de nodos de la red sobrepuesta.
4. Cada superpunto no debe necesitar servir a más de un número fijo de nodos normales.

Por fortuna, estos requerimientos son relativamente fáciles de cumplir en la mayoría de los sistemas punto a punto, dado que la red sobrepuesta es estructurada (como en los sistemas basados en DHT), o aleatoriamente no estructurada (como, por ejemplo, puede notarse en las soluciones basadas en el gossip). Demos un vistazo a las soluciones propuestas por Lo y colaboradores (2005).

En el caso de los sistemas basados en DHT, la idea básica es reservar una fracción del espacio identificador para los superpuntos. Recuerde que en los sistemas basados en DHT cada nodo recibe un identificador de m bits que es asignado aleatoria y uniformemente. Ahora suponga que reservamos los primeros k bits (es decir, los que están más a la izquierda) para identificar superpuntos. Por ejemplo, si necesitamos N superpuntos, entonces los primeros $[\log_2(N)]$ bits de cualquier *clave* puede utilizarse para identificar a estos nodos.

Para comprender lo anterior, supongamos que tenemos un (pequeño) sistema de cuerdas con $m = 8$ y $k = 3$. Cuando buscamos al nodo responsable de una clave específica p, podemos decidir enrutar primero la petición de búsqueda hacia el nodo responsable del patrón

p AND 11100000

el cual después se trata como el superpunto. Observe que cada *id* de nodo puede verificar si es un superpunto buscando

id AND 11100000

para ver si esta petición es enrutada hacia él mismo. Debido a que los identificadores de los nodos les son asignados uniformemente, puede verse que con un total de N nodos el número de superpuntos es, en promedio, igual a $2^{k-m}N$.

Un método completamente distinto se basa en el posicionamiento de los nodos en un espacio geométrico m-dimensional, como explicamos antes. En este caso, suponga que necesitamos colocar *uniformemente* N superpuntos a través de la sobrepuesta. La idea básica es simple: un total de N tokens se propagan por los N nodos elegidos al azar. Ningún nodo puede mantener más de un token. Cada token representa una fuerza de rechazo por lo que otro token prefiere retirarse. El efecto neto es que, si todos los token ejercen la misma fuerza de repulsión, se alejarán unos de otros y se propagarán uniformemente en el espacio geométrico.

Este método requiere que los nodos que tienen un token aprendan sobre otros token. Con esta finalidad, Lo y colaboradores proponen el uso de un protocolo de gossiping mediante el cual la fuerza de un token se disemina a través de la red. Si un nodo descubre que las fuerzas totales que actúan sobre él exceden un umbral, moverá el token en la dirección de fuerzas combinadas tal como lo muestra la figura 6-23.

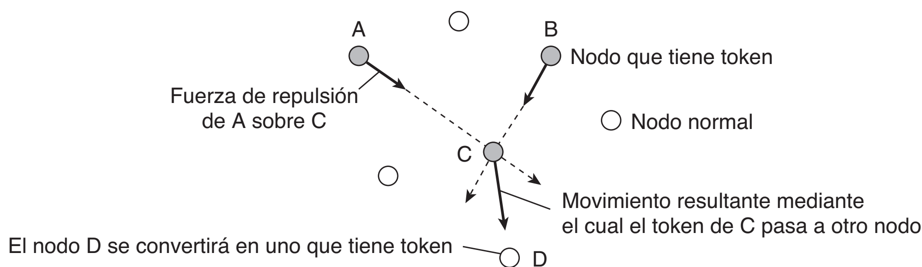


Figura 6-23. Tokens en movimiento en un espacio bidimensional que utilizan fuerzas de repulsión.

Cuando un token es mantenido por un nodo durante cierto tiempo, ese nodo se promoverá como superpunto.

6.6 RESUMEN

Muy relacionada con la comunicación entre procesos se encuentra la cuestión de cómo sincronizar procesos en los sistemas distribuidos. La sincronización trata de hacer lo correcto en el momento correcto. Un problema de los sistemas distribuidos, y de las redes de computadoras en general, es que no hay un concepto de un reloj globalmente compartido. En otras palabras, los procesos presentes en diferentes máquinas tienen su propio concepto de lo que es el tiempo.

Hay varias maneras de sincronizar los relojes de un sistema distribuido, pero todos los métodos se basan esencialmente en el intercambio de valores de reloj, mientras se toma en cuenta el tiempo que toma el envío y la recepción de mensajes. Las variaciones en los retrasos de comunicación y la forma en que dichas variaciones se manejan determinan de manera importante la precisión de los algoritmos de sincronización de reloj.

Relacionado con estos problemas de sincronización se encuentra el posicionamiento de los nodos en una sobrepuesta geométrica. La idea básica es asignar coordenadas a cada nodo, a partir de un espacio m-dimensional, tal que la distancia geométrica pueda utilizarse como una medida exacta para la latencia entre dos nodos. El método de asignación de coordenadas se parece mucho al aplicado para determinar la ubicación y el tiempo en un GPS.

En muchos casos, saber el tiempo absoluto no es necesario. Lo que cuenta es que los eventos relacionados en diferentes procesos ocurran en el orden correcto. Lamport mostró que introduciendo la idea de los relojes lógicos es posible que una colección de procesos logre la coincidencia global sobre el ordenamiento correcto de eventos. En esencia, a cada evento e, tal como enviar o recibir un mensaje, se le asigna un registro de tiempo lógico y globalmente único, $C(e)$, tal que cuando el evento a ocurre antes que el b, $C(a) < C(b)$. Los registros de tiempo de Lamport pueden extenderse a registros vectoriales: si $C(a) < C(b)$, incluso sabemos que el evento a causalmente precedió a b.

Una importante clase de algoritmos de sincronización es la de la exclusión mutua distribuida. Estos algoritmos garantizan que, en una colección de procesos distribuidos, al menos un proceso tenga en cierto momento acceso a un recurso compartido. La exclusión mutua distribuida puede lograrse fácilmente si utilizamos un coordinador que rastree el turno de cada proceso. Los algoritmos completamente distribuidos también existen, pero tienen la desventaja de que son más susceptibles a fallas de comunicación y de procesos.

La sincronización entre procesos con frecuencia requiere que uno de los procesos actúe como coordinador. En los casos donde el coordinador no es fijo, es necesario que los procesos de un cálculo distribuido decidan qué proceso será el coordinador. Tal decisión se toma mediante algoritmos de elección. Los algoritmos de elección se utilizan básicamente en los casos donde el coordinador puede fallar. Sin embargo, también pueden aplicarse para la selección de superpuntos en sistemas punto a punto.

PROBLEMAS

1. Mencione al menos tres fuentes de retraso que pueden introducirse entre transmisiones WWV del tiempo y los procesadores de un sistema distribuido configurando sus relojes internos.

2. Considere el comportamiento de dos máquinas en un sistema distribuido. Ambas tienen relojes que se supone deben hacer marcas 1000 veces por milisegundo. Una máquina sí lo hace, pero la otra hace marcas sólo 990 veces por milisegundo. Si las actualizaciones UTC llegan una vez por minuto, ¿cuál es el máximo desajuste de reloj que ocurrirá?

3. Unos de los dispositivos modernos que han llegado (silenciosamente) a los sistemas distribuidos son los receptores GPS. Proporcione ejemplos de aplicaciones distribuidas que pueden utilizar la información GPS.

4. Cuando un nodo sincroniza su reloj con el de otro nodo, en general es buena idea considerar mediciones anteriores. ¿Por qué? Además, proporcione un ejemplo sobre cómo las lecturas anteriores pueden tomarse en cuenta.

5. Agregue un nuevo mensaje a la figura 6-9 que sea concurrente con el mensaje *A*, es decir, que no ocurra antes que *A* ni después que *A*.

6. Para lograr una transmisión totalmente ordenada con registros de tiempo Lamport, ¿es estrictamente necesario que cada mensaje sea acusado de recibido?

7. Considere una capa de comunicación en la que los mensajes se entreguen sólo en el orden en que se enviaron. Proporcione un ejemplo donde incluso este ordenamiento es innecesariamente restrictivo.

8. Muchos algoritmos distribuidos requieren el uso de un proceso coordinador. ¿En qué medida pueden tales algoritmos considerarse en realidad distribuidos? Explique su respuesta.

9. En el método centralizado de exclusión mutua (figura 6-14), una vez que se recibe un mensaje de un proceso que libera su acceso exclusivo a los recursos que estaba utilizando el coordinador normalmente otorga su permiso al primer proceso de la cola. Proporcione otro posible algoritmo para el coordinador.

10. Considere nuevamente la figura 6-14. Suponga que el coordinador falla. ¿Esto siempre provoca que el sistema se caiga? Si no es así, ¿bajo qué circunstancias ocurre la caída? ¿Existe alguna manera de evitar el problema y hacer que el sistema tolere la falla del coordinador?

11. El algoritmo de Ricart y Agrawala tiene el problema de que si un proceso ha fallado y no responde una petición de otro proceso para acceder a los recursos, la falta de respuesta se interpretará como una negativa de permiso. Nosotros sugerimos que todas las peticiones sean contestadas de inmediato para facilitar la detección de procesos fallidos. ¿Existe alguna circunstancia en la que incluso este método resulte insuficiente? Explique su respuesta.

12. ¿Cómo cambian las entradas de la figura 6-17 si suponemos que los algoritmos pueden implementarse en una LAN que soporte transmisiones de hardware?

13. Un sistema distribuido puede tener varios recursos independientes. Imagine que el proceso 0 quiere acceso al recurso *A*, y el proceso 1 quiere acceso al recurso *B*. ¿El algoritmo de Ricart y Agrawala puede ocasionar puntos muertos? Explique su respuesta.

14. Suponga que dos procesos detectan simultáneamente la desactivación del coordinador, y ambos deciden celebrar una elección utilizando el algoritmo del abusón (Bully). ¿Qué ocurre entonces?

15. En la figura 6-21 tenemos dos mensajes de *ELECCIÓN* circulando simultáneamente. Ya que tener dos de estos mensajes no ocasiona daños, resultaría más elegante si pudiera eliminarse uno. Plantee un algoritmo que haga esto sin afectar la operación de la elección básica del algoritmo.

16. (**Asignación para el laboratorio.**) Los sistemas UNIX proporcionan muchas facilidades para mantener las computadoras en sincronía, de manera notable la combinación de la herramienta *crontab* (la cual permite agendar automáticamente las operaciones) y varios comandos de sincronización resulta poderosa. Configure un sistema UNIX que mantenga exacto el tiempo local dentro del rango de un solo segundo. De igual forma, configure una herramienta automática de respaldo mediante la que cierto número de archivos cruciales se transfieran automáticamente a una máquina remota una vez cada 5 minutos. Su solución debe ser eficiente con respecto al uso del ancho de banda.

7

CONSISTENCIA Y REPLICACIÓN

Una cuestión importante en los sistemas distribuidos es la replicación de datos. Por lo general, los datos se replican para incrementar la confiabilidad o mejorar el rendimiento. Uno de los principales problemas es hacer que las réplicas se mantengan consistentes. De modo informal, esto significa que cuando se actualiza una copia, necesitamos garantizar que las demás copias también se actualicen; de otra manera las réplicas dejarán de ser lo mismo. En este capítulo veremos detalladamente lo que significa en realidad la consistencia de datos replicados, y las diferentes formas de lograr esa consistencia.

Iniciamos con una introducción general que explica por qué la replicación es útil, y cómo se relaciona con la escalabilidad. Después continuamos con lo que significa en realidad la consistencia. Una clase importante de lo que conocemos como modelos de consistencia supone que varios procesos acceden simultáneamente a datos compartidos. En estas situaciones, la consistencia puede formularse con respecto a lo que los procesos pueden esperar cuando leen y actualizan los datos compartidos, sabiendo que otros procesos también acceden a esos datos.

Los modelos de consistencia para datos compartidos con frecuencia resultan difíciles de implementar en sistemas distribuidos a gran escala. Además, en muchos casos es posible utilizar modelos más simples, los cuales también son más fáciles de implementar. Una clase específica está formada por los modelos de consistencia centrados en el cliente, los cuales se concentran en la consistencia desde la perspectiva de un solo cliente (posiblemente móvil). En una sección aparte, explicaremos los modelos de consistencia centrados en el cliente.

La consistencia representa sólo la mitad de la historia. También debemos considerar cómo se implementa. Existen básicamente dos cuestiones, más o menos independientes, que debemos tener

presentes. Primero que todo, iniciaremos concentrándonos en la administración de réplicas, lo cual toma en cuenta no sólo la ubicación de los servidores de réplicas, sino también cómo se distribuye el contenido a estos servidores.

El segundo asunto es cómo se mantienen consistentes las réplicas. En la mayoría de los casos, las aplicaciones requieren una forma fuerte de consistencia. De modo informal esto significa que las actualizaciones se propagarán de manera más o menos inmediata entre las réplicas. Existen varias alternativas para implementar una consistencia fuerte, las cuales explicaremos en una sección aparte. Además veremos protocolos de cacheo, los cuales constituyen un caso especial de protocolos de consistencia.

7.1 INTRODUCCIÓN

En esta sección, iniciamos explicando las importantes razones que existen para desear la replicación de datos. Nos concentramos en la replicación como una técnica útil para lograr la escalabilidad, y en comprender por qué el razonamiento sobre la consistencia es tan importante.

7.1.1 Razones para la replicación

Existen dos razones principales para replicar datos: la confiabilidad y el rendimiento. Primero, los datos se replican para incrementar la confiabilidad de un sistema. Si un sistema de archivos se replicó, es posible continuar trabajado después de que una réplica falle con tan sólo cambiar a una de las otras réplicas. Además, al mantener varias copias se hace posible proporcionar una mejor protección contra datos corruptos. Por ejemplo, imagine que hay tres copias de un archivo y que cada operación de lectura y escritura se realiza en cada copia. Podemos protegernos contra una operación de escritura defectuosa, si consideramos que el valor devuelto por al menos dos copias es el correcto.

La otra razón para replicar datos es el rendimiento. La replicación es importante para el rendimiento cuando el sistema distribuido necesita escalar en números y en área geográfica. Por ejemplo, el escalamiento en números ocurre cuando un número creciente de procesos necesita acceder a datos que son administrados por un solo servidor. En ese caso, el rendimiento puede mejorarse replicando el servidor y, posteriormente, dividiendo el trabajo.

Escalar con respecto al tamaño de un área geográfica también puede requerir de la replicación. La idea básica es que al colocar una copia de los datos en la proximidad del proceso que los usa, el tiempo de acceso a los datos disminuye. En consecuencia, el rendimiento percibido por ese proceso aumenta. Este ejemplo también muestra que puede ser difícil evaluar los beneficios de la replicación en cuanto al rendimiento. Aunque un proceso cliente puede percibir un mejor rendimiento, también puede darse el caso de que se consuma más ancho de banda de la red para mantener todas las réplicas actualizadas.

Si la replicación ayuda a mejorar la confiabilidad y el rendimiento, ¿quién estaría en su contra? Por desgracia, hay un precio a pagar cuando se replican datos. El problema con la replicación es que tener muchas copias puede provocar problemas de consistencia. Siempre que se modifica una copia, ésta se vuelve diferente al resto de copias. Por tanto, para garantizar la consistencia, las modificaciones deben realizarse en todas las copias. El precio de la replicación lo determinan exactamente el cuándo y el cómo deben realizarse dichas modificaciones.

Para comprender el problema, considere el mejorar los tiempos de acceso a páginas web. Si no se toman medidas especiales, en ocasiones solicitar una página a un servidor web remoto puede incluso tomar varios segundos. Para mejorar el rendimiento, los navegadores web almacenan localmente una copia de una página previamente solicitada (es decir, **cachean** una página web). Si un usuario requiere nuevamente esa página, el navegador devuelve automáticamente la copia local. El tiempo de acceso percibido por el usuario es excelente. Sin embargo, si el usuario siempre quiere la versión más reciente de una página, podría tener mala suerte. El problema es que si la página se modificó entretanto, las modificaciones no se habrán propagado a las copias cacheadas, lo cual hará que esas copias no estén actualizadas.

Una solución para el problema de devolver una copia vieja al usuario es, en primer lugar, prohibir al navegador mantener copias locales, y dejar que el servidor se encargue totalmente de la replicación. Sin embargo, esta solución puede ocasionar tiempos de acceso deficientes si no se coloca alguna réplica cerca del usuario. Otra solución es dejar que el servidor web invalide o actualice cada copia cacheada, pero eso requiere que el servidor dé seguimiento a todos los cachés y les envíe mensajes. Esto, a su vez, puede degradar todo el rendimiento del servidor. Más adelante retomaremos las cuestiones de rendimiento *versus* escalabilidad.

7.1.2 Replicación como técnica de escalamiento

Replicación y cacheo para el rendimiento se utilizan ampliamente como técnicas de escalamiento. Las cuestiones de escalabilidad generalmente aparecen en forma de problemas de rendimiento. Colocar copias de datos cerca de los procesos que los utilizan puede mejorar el rendimiento mediante la reducción del tiempo de acceso, y resolver así los problemas de escalabilidad.

Una compensación necesaria es que para mantener copias actualizadas se requiere un mayor ancho de banda en la red. Considere un proceso P que accede a una réplica local N veces por segundo, mientras que a la réplica se le actualiza M veces por segundo. Suponga que una actualización refresca completamente la versión anterior de la réplica local. Si $N << M$, es decir, la velocidad de acceso a la actualización es muy baja, se presenta la situación en que el proceso P nunca accede a muchas versiones actualizadas de la réplica local, ello se traduce en que la comunicación de la red sea inútil para esas versiones. En este caso, tal vez hubiese sido mejor no instalar una réplica local cerca de P, o aplicar una estrategia diferente para actualizar la réplica. Retomaremos estas cuestiones más adelante.

Sin embargo, un problema todavía más serio es que mantener varias copias consistentes puede, por sí mismo, estar sujeto a serios problemas de escalabilidad. Por intuición, una colección de

copias es consistente cuando las copias siempre son las mismas. Esto significa que una operación de lectura realizada a cualquier copia siempre devolverá el mismo resultado. En consecuencia, cuando a una copia se le realice una operación de actualización, dicha actualización debe propagarse a todas las copias antes de que ocurra una operación posterior, independientemente de en qué copia se inicie o realice esa operación.

A este tipo de consistencia algunas veces se le denomina, de manera informal (e imprecisa), consistencia hermética, como en el caso de la llamada replicación sincrónica. (En la siguiente sección daremos definiciones precisas de consistencia, y presentaremos una gama de modelos de consistencia.) La idea principal es que una actualización se realiza en todas las copias como una sola operación atómica, o transacción. Por desgracia, implementar atomicidad involucrando una gran cantidad de réplicas, que pueden estar ampliamente dispersas a través de una red de gran escala, es inherentemente difícil cuando se necesita completar operaciones rápidamente.

Las dificultades surgen del hecho de que necesitamos sincronizar todas las réplicas. En esencia, esto significa que todas las réplicas primero necesitan acordar exactamente cuándo se realizará una actualización local. Por ejemplo, las réplicas pueden necesitar decidir un ordenamiento global de operaciones, utilizando registros de tiempo Lamport, o dejar que un coordinador asigne el orden. La sincronización global simplemente requiere mucho tiempo de comunicación, en especial cuando las réplicas se dispersan a través de una red de área amplia.

Ahora enfrentamos un dilema. Por una parte, los problemas de escalabilidad pueden disminuirse aplicando replicación y cacheo, lo que deriva en un mejor rendimiento. Por otra, mantener consistentes a todas la copias generalmente, requiere de una sincronización global, y ello es inherentemente costoso en términos de rendimiento. La cura puede resultar peor que la enfermedad.

En muchos casos, la única solución real es disminuir las restricciones de consistencia. En otras palabras, si podemos relajar el requerimiento de que las actualizaciones necesitan ejecutarse como operaciones atómicas, tal vez podamos evitar las sincronizaciones globales (instantáneas), y quizá aumentar así el rendimiento. El precio a pagar es que las copias podrían no ser las mismas en todas partes. Como es evidente, hasta dónde relajar la consistencia depende en gran medida de los patrones de acceso y actualización de los datos replicados, así como del propósito de utilizar esos datos.

En las siguientes secciones, primero consideramos una gama de modelos de consistencia, y proporcionamos definiciones precisas sobre lo que realmente significa consistencia. Después continuamos con una explicación sobre diferentes formas de implementar estos modelos, a través de lo que se conoce como protocolos de distribución y consistencia. Diferentes métodos para clasificar la consistencia y la replicación pueden encontrarse en Gray y colaboradores (1996), y en Wiesmann y colaboradores (2000).

7.2 MODELOS DE CONSISTENCIA CENTRADA EN LOS DATOS

De manera tradicional, la consistencia se ha explicado en el contexto de operaciones de lectura y escritura sobre datos compartidos, disponibles mediante memoria compartida (distribuida), una base de datos compartida (distribuida), o un sistema de archivo (distribuido). En esta sección, uti-

lizamos el término más amplio denominado **almacén de datos**. Un almacén de datos puede estar físicamente distribuido en varias máquinas. En particular, se asume que todo proceso que puede acceder a datos del almacén tiene una copia local (o en las cercanías) disponible de todo el almacén. Las operaciones de escritura se propagan hacia las otras copias, como se muestra en la figura 7-1. Una operación de datos se clasifica como una operación de escritura cuando ésta cambia los datos, de otro modo se clasifica como una operación de lectura.

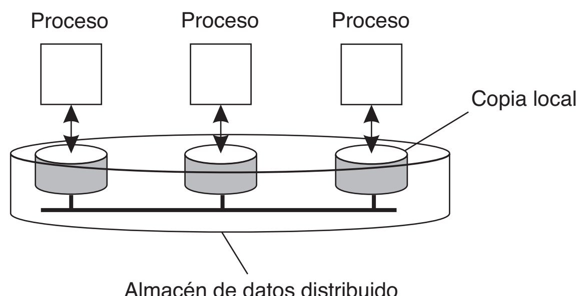


Figura 7-1. Organización general de un almacén de datos lógicos, físicamente distribuido y replicado a través de múltiples procesos.

Un **modelo de consistencia** es básicamente un contrato entre los procesos y el almacén de datos. Este contrato dice que si los procesos aceptan obedecer ciertas reglas, el almacén promete funcionar correctamente. En general, un proceso que realiza una operación de lectura sobre un elemento de datos espera que la operación devuelva un valor que muestre los resultados de la última operación de escritura sobre los datos.

En la ausencia de un reloj global, es difícil definir precisamente cuál es la última operación de escritura. Como alternativa, necesitamos proporcionar otras definiciones, lo que nos lleva a una gama de modelos de consistencia. Cada modelo restringe efectivamente los valores que puede devolver una operación de lectura sobre un elemento de datos. Como es de esperarse, los modelos con más restricciones son más fáciles de utilizar, por ejemplo, cuando se desarrollan aplicaciones, mientras que aquellos con menos restricciones resultan más difíciles. La desventaja es, por supuesto, que los modelos fáciles de usar no se desempeñan tan bien como los difíciles. Así es la vida.

7.2.1 Consistencia continua

De lo que hemos explicado hasta ahora, debe resultar claro que no hay algo que pueda considerarse como la mejor solución para replicar datos. La replicación de datos posee problemas de consistencia que no pueden resolverse eficientemente de una forma general. Sólo si relajamos la consistencia podemos tener la esperanza de encontrar soluciones eficientes. Por desgracia, tampoco existen reglas generales para relajar la consistencia: exactamente lo que puede tolerarse depende, en gran medida, de las aplicaciones.

Hay diferentes formas en que las aplicaciones especifican las inconsistencias que pueden tolerar. Yu y Vahdat (2002) consideran un método general para diferenciar tres ejes independientes para definir inconsistencias: desviación en valores numéricos entre réplicas, desviación en el deterioro entre réplicas, y desviación con respecto al ordenamiento de operaciones de actualización. Yu y Vahdat se refieren a estas desviaciones como rangos de **consistencia continua**.

Medir la inconsistencia en términos de desviaciones numéricas puede utilizarse en aplicaciones para las que los datos tienen semánticas numéricas. Un ejemplo evidente es la replicación de registros que contienen precios de acciones. En este caso, una aplicación puede especificar que dos copias no deben desviarse más de $0.02, lo cual sería una *desviación numérica absoluta*. Como alternativa, podría especificarse una *desviación numérica relativa*, lo cual establece que dos copias deben diferir no más de, por ejemplo, 0.5%. En ambos casos, veríamos que si una acción va hacia arriba (y una de las réplicas se actualiza inmediatamente) sin violar las desviaciones numéricas especificadas, las réplicas aún serían consideradas como mutuamente consistentes.

La desviación numérica también puede comprenderse en términos del número de actualizaciones que se han aplicado a una réplica dada, pero que aún no han sido vistas por otras réplicas. Por ejemplo, un caché web puede no haber visto un lote de operaciones realizadas por un servidor web. En este caso, la desviación asociada con el *valor* también se conoce como su *ponderación*.

Las desviaciones viejas se relacionan con la última vez que se actualizó una réplica. Para algunas aplicaciones, es tolerable que una réplica proporcione datos viejos siempre y cuando no sean *demasiado* viejos. Por ejemplo, los informes sobre el clima permanecen a menudo razonablemente precisos durante cierto tiempo, digamos algunas horas. En tales casos, un servidor principal puede recibir actualizaciones oportunas, pero decidir propagar las actualizaciones a las réplicas de vez en cuando.

Por último, hay clases de aplicaciones en las que se permite que el ordenamiento de actualizaciones sea diferente en varias réplicas, siempre que las diferencias sean limitadas. Una manera de ver estas actualizaciones es que se aplican tentativamente a una copia local, en espera de un acuerdo global de todas las réplicas. En consecuencia, algunas actualizaciones necesitarán repetirse y tendrán que aplicarse en un orden diferente antes de volverse permanentes. Por intuición, el ordenamiento de desviaciones es mucho más difícil de comprender que las otras dos métricas de consistencia. Más adelante proporcionaremos ejemplos que clarificarán las cosas.

La idea de una conit

Para definir inconsistencias, Yu y Vahdat presentaron una unidad de consistencia, abreviada como **conit**. Una conit especifica la unidad con la que se medirá la consistencia. Así, en nuestro ejemplo de intercambio de acciones, una conit podría definirse como un registro que representa una sola acción. Otro ejemplo es un informe individual del clima.

Para dar un ejemplo de una conit, y al mismo tiempo ilustrar las desviaciones numérica y de ordenamiento, considere las dos réplicas que muestra la figura 7-2. Cada réplica i mantiene un reloj vectorial bidimensional, VC_i, como los relojes descritos en el capítulo 6. Utilizamos la notación t, i para expresar una operación que fue realizada por la réplica i en (su) tiempo lógico t.

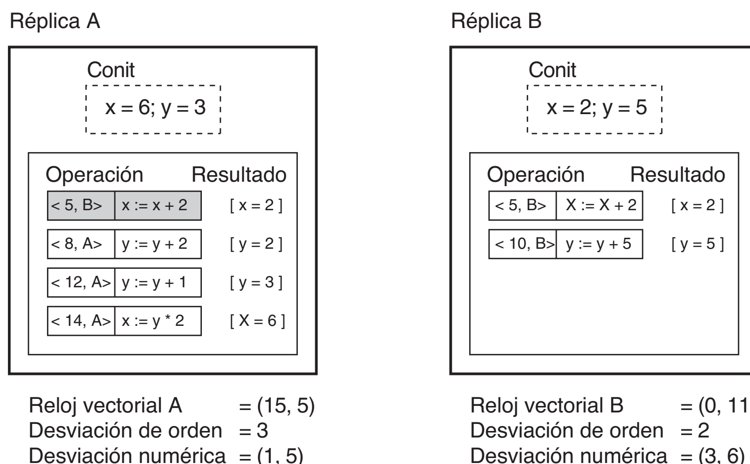


Figura 7-2. Ejemplo sobre cómo dar seguimiento a desviaciones de consistencia [adaptado de (Yu y Vahdat, 2002)].

En este ejemplo vemos dos réplicas que operan en una conit que contiene los elementos de datos *x* y *y*. Suponemos que ambas variables han sido inicializadas en 0. La réplica A recibió la operación

$$5,B : x \leftarrow x + 2$$

de la réplica *B*, y la ha vuelto permanente (es decir, la operación ha sido confirmada en *A* y no puede deshacerse). La réplica *A* tiene tres operaciones tentativas de actualización: 8,*A*, 12,*A*, y 14,*A*, las cuales llevan a su desviación de ordenamiento a 3. También advierta que debido a la última operación, 14,A, el reloj vectorial de A se vuelve (15,5).

La única operación de *B* que *A* aún no ha visto es 10,*B*, llevando su desviación numérica a 1 con respecto a las operaciones. En este ejemplo, la ponderación de esta desviación puede expresarse como la diferencia máxima entre los valores (confirmados) de *x* y *y* en *A*, y el resultado de las operaciones en *B* no vistas por *A*. El valor confirmado en *A* es $(x,y) = (2,0)$, mientras que la operación en *B*, no vista por *A*, arroja una diferencia de $y = 5$.

Un razonamiento similar muestra que *B* tiene dos operaciones tentativas de actualización: 5,*B* y 10,*B*, lo cual significa que tiene una desviación de ordenamiento de 2. Debido a que *B* aún no ha visto una sola operación de *A*, su reloj vectorial se vuelve (0,11). La desviación numérica es 3 con una ponderación total de 6. Este último valor proviene del hecho de que el valor confirmado de *B* es $(x,y) = (0,0)$, mientras que las operaciones tentativas en *A* ya llevarán 6 en *x*.

Observe que hay ventajas y desventajas entre mantener conits de granularidad fina y conits de granularidad gruesa. Si una conit representa muchos datos, tal como una base de datos completa, entonces las actualizaciones se aplican a todos los datos incluidos en la conit. En consecuencia, esto puede hacer que las réplicas entren más rápido en un estado de inconsistencia. Por ejemplo, suponga

que en la figura 7-3 dos réplicas pueden diferir en no más de una actualización pendiente. En ese caso, cuando cada uno de los elementos de datos de la figura 7-3(a) han sido actualizados una vez en la primera réplica, la segunda réplica también necesitará actualizarse. Éste no es el caso cuando se elige una conit más pequeña, como nos muestra la figura 7-3(b). Ahí, las réplicas aún se consideran actualizadas. En particular, este problema resulta importante cuando los elementos de datos contenidos en una conit se utilizan en forma completamente independiente, en cuyo caso se dice que **comparten falsamente** la conit.

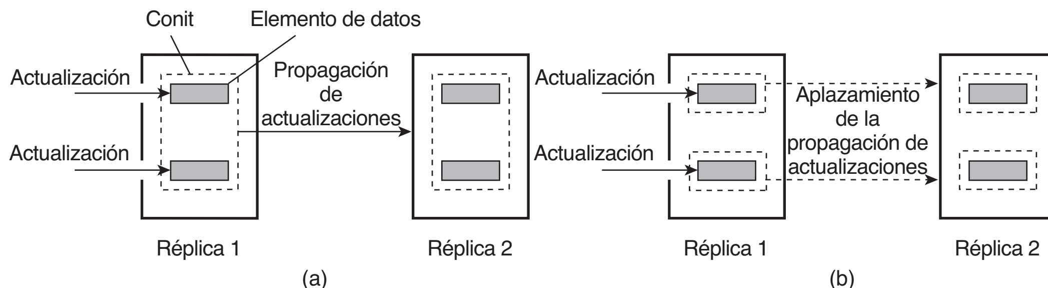


Figura 7-3. Elección de la granularidad adecuada para una conit. (a) Dos actualizaciones dan pie a la propagación de actualizaciones. (b) Ninguna actualización es necesaria (aún).

Por desgracia, implementar conits muy pequeñas no es buena idea, por la sencilla razón de que el número total de conits que se necesita manejar también aumenta. En otras palabras, hay una sobrecarga relacionada con el manejo de conits que deben tomarse en cuenta. Esta sobrecarga, a su vez, puede afectar de modo adverso todo el rendimiento, lo cual debe tomarse en cuenta.

Aunque desde un punto de vista conceptual las conits forman un medio atractivo para capturar los requerimientos de consistencia, hay dos puntos importantes con los que debemos tratar antes de poder ponerlas en práctica. Primero, para reforzar la consistencia necesitamos protocolos. Más adelante en este capítulo explicaremos dichos protocolos.

Un segundo punto es que los desarrolladores de programas deben especificar los requerimientos de consistencia necesarios para sus aplicaciones. La práctica indica que obtener tales requerimientos puede resultar extremadamente difícil. Por lo general, los programadores no acostumbran manejar la replicación, y dejan a un lado lo que significa proporcionar información detallada sobre consistencia. Por tanto, es muy importante que existan interfaces de programación sencillas y fáciles de entender.

La consistencia continua puede implementarse como un conjunto de herramientas que aparezca ante los programadores tan sólo como otra biblioteca a vincular con sus aplicaciones. Una conit simplemente se declara al lado de una actualización de un elemento de datos. Por ejemplo, el fragmento de seudocódigo

```
AfectaConit(ConitQ, 1, 1);
agrega el mensaje m a la cola Q;
```

establece que agregar un mensaje a la cola *Q* pertenece a una conit llamada "ConitQ". Asimismo, las operaciones ahora pueden declararse como dependientes de las conits:

```
DependeDeLaConit(ConitQ, 4, 0, 60);
lee el mensaje m de la cabeza de la cola Q;
```

En este caso, la llamada a `DependeDeLaConit()` especifica que la desviación numérica, la desviación de ordenamiento, y la desviación vieja deben limitarse a los valores 4, 0, y 60 (segundos), respectivamente. Esto puede interpretarse como que debe haber cuando mucho 4 operaciones de actualización no vistas en otras réplicas, que no debe haber actualizaciones locales tentativas, y que la antigüedad de la copia local de *Q* debe haberse verificado hace no más de 60 segundos. Si no se satisfacen estos requerimientos, el middleware subyacente intentará llevar la copia local de *Q* a un estado tal que la operación de lectura pueda llevarse a cabo.

7.2.2 Ordenamiento consistente de operaciones

Además de consistencia continua, desde la década pasada hay un gran cuerpo de trabajo dedicado a modelos de consistencia centrada en datos. Una clase importante de modelos proviene del campo de programación concurrente. Confrontados con el hecho de que en la computación paralela y distribuida varios procesos necesitarán compartir recursos y acceder simultáneamente a ellos, los investigadores han buscado expresar la semántica de accesos concurrentes cuando se replican recursos compartidos. Esto ha derivado en al menos un importante modelo de consistencia que se utiliza ampliamente. A continuación nos concentraremos en lo que se conoce como consistencia secuencial, y también explicaremos una variante más débil, llamada consistencia causal.

Los modelos que explicamos en esta sección tratan con operaciones de ordenamiento consistente sobre datos compartidos y replicados. En principio, los modelos superan a aquellos de consistencia continua en el sentido de que cuando es necesario confirmar actualizaciones en réplicas, éstas tendrán que acordar un ordenamiento global de esas actualizaciones. En otras palabras, necesitan acordar un ordenamiento consistente de esas actualizaciones. Los modelos de consistencia que explicaremos a continuación tratan sobre cómo lograr tales ordenamientos consistentes.

Consistencia secuencial

A continuación, utilizaremos una notación especial en la que trazaremos las operaciones de un proceso a lo largo de un eje del tiempo. El eje del tiempo siempre se traza horizontalmente, y aumenta de izquierda a derecha. Los símbolos

$W_i(x)a$ y $R_i(x)b$

significan que se han realizado, respectivamente, la escritura del proceso P_i sobre el elemento de datos x con el valor a y una lectura de ese elemento por P_i devolviendo b. Suponemos que cada elemento de datos es inicialmente *NIL*. Cuando no hay confusión con respecto a qué proceso está accediendo a los datos, omitimos el subíndice de los símbolos W y R.



Figura 7-4. Comportamiento de dos procesos operando sobre el mismo elemento de datos. El eje horizontal representa el tiempo.

Como ejemplo, en la figura 7-4 P_1 realiza una escritura en el elemento de datos *x*, modificando su valor para *a*. Observe que, en principio, esta operación, $W_1(x)a$, se realiza primero sobre una copia del almacén de datos que es local para P_1, y después se propaga hacia otras copias locales. En nuestro ejemplo, P_2 lee después el valor *NIL*, y un tiempo después lee *a* (desde su copia local del almacén). Lo que vemos aquí es que lleva cierto tiempo propagar la actualización de *x* hacia P_2, lo cual es perfectamente aceptable.

La **consistencia secuencial** es un importante modelo de consistencia centrado en los datos, el cual fue definido por primera vez por Lamport (1979) en el contexto de memoria compartida para sistemas de multiprocesador. En general, se dice que un almacén de datos es secuencialmente consistente cuando satisface la siguiente condición:

El resultado de cualquier ejecución es el mismo que si las operaciones (de lectura y escritura) de todos los procesos efectuados sobre el almacén de datos se ejecutaran en algún orden secuencial y las operaciones de cada proceso individual aparecieran en esa secuencia en el orden especificado por su programa.

Lo que esta definición significa es que, cuando los procesos se ejecutan concurrentemente en (quizá) diferentes máquinas, cualquier interpolación válida de operaciones de lectura y escritura es un comportamiento aceptable, pero *todos los procesos ven la misma interpolación de operaciones*. Observe que nada se dice sobre el tiempo; es decir, no hay referencia a la operación de escritura "más reciente" sobre el elemento de datos. Advierta que, en este contexto, un proceso "ve" escrituras de todos los procesos, pero sólo ve sus propias lecturas.

Que el tiempo no juega un papel importante puede verse en la figura 7-5. Consideremos cuatro procesos operando sobre el mismo elemento de datos *x*. En la figura 7-5(a), el proceso P_1 primero realiza $W(x)a$ para *x*. Después (en tiempo absoluto), el proceso P_2 también realiza una operación de escritura, al establecer el valor de *x* para *b*. Sin embargo, los procesos P_3 y P_4 *primero* leen el valor *b*, y después el valor *a*. En otras palabras, la operación de escritura del proceso P_2 parece haber ocurrido antes que la de P_1.

Por contraste, la figura 7-5(b) viola la consistencia secuencial ya que no todos los procesos ven la misma interpolación de operaciones de escritura. En particular, para el proceso P_3, parece como si el elemento de datos primero hubiese sido cambiado a *b*, y después a *a*. Por otra parte, P_4 concluirá que el valor final es *b*.

Para concretar más la idea de consistencia secuencial, consideremos los tres procesos, P_1, P_2, y P_3, en ejecución concurrente, que aparecen en la figura 7-6 (Dubois y cols., 1988). Los elementos de datos de este ejemplo están formados por las tres variables enteras *x*, *y*, y *z*, y están guardados en un almacén de datos (posiblemente distribuido) compartido y secuencialmente consistente.

P1: W(x)a				
P2:	W(x)b			
P3:		R(x)b		R(x)a
P4:			R(x)b	R(x)a

(a)

P1: W(x)a				
P2:	W(x)b			
P3:		R(x)b		R(x)a
P4:			R(x)a	R(x)b

(b)

Figura 7-5. (a) Almacén de datos secuencialmente consistente. (b) Almacén de datos que no es secuencialmente consistente.

Proceso 1	**Proceso 2**	**Proceso 3**
x ← 1; impresión (y, z)	y ← 1; impresión (x, z)	z ← 1; impresión (x, y)

Figura 7-6. Tres procesos en ejecución concurrente.

Suponemos que cada variable se inicializa en 0. En este ejemplo, una asignación corresponde a una operación de escritura, mientras que una instrucción de impresión corresponde a una operación simultánea de lectura de sus dos argumentos. Suponemos que todas las instrucciones son indivisibles.

Varias secuencias de ejecución interpoladas son posibles. Con seis instrucciones independientes, potencialmente existen 720 (6!) secuencias de ejecución posibles, aunque algunas violan el orden del programa. Consideremos las 120 (5!) secuencias que comienzan con $x \leftarrow 1$. La mitad tiene *print*(x,z) antes que $y \leftarrow 1$ y viola el orden del programa. La mitad también tiene *print* (x,y) antes que $z \leftarrow 1$, y también violan el orden del programa. Sólo una cuarta parte de las 120 secuencias, o 30, son válidas. Otras 30 secuencias válidas son posibles empezando con $y \leftarrow 1$ y 30 más pueden comenzar con $z \leftarrow 1$, para un total de 90 secuencias de ejecución válidas. Cuatro de éstas aparecen en la figura 7-7.

En la figura 7-7(a), los tres procesos están en orden de ejecución, primero P_1, luego P_2, y después P_3. Los otros tres ejemplos muestran diferentes, pero igualmente válidas, interpolaciones de las instrucciones en el tiempo. Cada uno de los tres procesos imprime dos variables. Debido a que los únicos valores que cada variable puede tomar son el valor inicial (0), o el valor asignado (1), cada proceso produce una cadena de 2 bits. Los números posteriores a las *Impresiones* son las salidas reales que aparecen en el dispositivo de salida.

Si concatenamos la salida de P_1, P_2, y P_3 en ese orden, obtenemos una cadena de 6 bits que caracteriza una interpolación particular de instrucciones. Esta cadena es la listada como *Firma* en la figura 7-7. Más adelante caracterizaremos cada ordenamiento mediante su firma, en lugar de hacerlo con su impresión.

No todos los 64 patrones de firma están permitidos. Como un ejemplo trivial, 000000 no está permitido, ya que implicaría que las instrucciones de impresión se ejecutaran antes que las instrucciones de asignación, lo cual viola el requerimiento de que las instrucciones se ejecutan en el orden del programa. Un ejemplo más sutil es 001001. Los primeros dos bits, 00, significan que y y z eran 0

x ← 1;	x ← 1;	y ← 1;	y ← 1;
impresión (y, z);	y ← 1;	z ← 1;	x ← 1;
y ← 1;	impresión (x, z);	impresión (x, y);	z ← 1;
impresión (x, z);	impresión (y, z);	impresión (x, z);	impresión (x, z);
z ← 1;	z ← 1;	x ← 1;	impresión (y, z);
impresión (x, y)	impresión (x, y)	impresión (y, z)	impresión (x, y);
Firma: 001011	Firma: 101011	Firma: 010111	Firma: 111111
Impresiones: 001011	Impresiones: 101011	Impresiones: 110101	Impresiones: 111111
(a)	(b)	(c)	(d)

Figura 7-7. Cuatro secuencias válidas de ejecución para los procesos de la figura 7-6. El eje vertical representa el tiempo.

cuando P_1 hizo su impresión. Esta situación sólo ocurre cuando P_1 ejecuta ambas instrucciones antes de que inicien P_2 o P_3. Los dos siguientes bits, 10, significan que P_2 debe ejecutarse después de que P_1 ha iniciado, pero antes de que P_3 haya iniciado. Los dos últimos bits, 01, significan que P_3 debe completarse antes de que P_1 inicie, pero ya vimos que P_1 debe ir primero. Por tanto, 001001 no está permitido.

En resumen, los 90 diferentes ordenamientos válidos producen una variedad de diferentes resultados de programa (aunque menos de 64) que son permitidos bajo la suposición de consistencia secuencial. El contrato entre los procesos y el almacén de datos compartido y distribuido es que el proceso debe aceptar todos estos resultados como válidos. En otras palabras, los procesos deben aceptar los cuatro resultados que aparecen en la figura 7-7 y todos los otros resultados válidos como respuestas apropiadas, y deben funcionar correctamente si cualquiera de ellos ocurre. Un programa que funciona con alguno de estos resultados y con otros no viola el contrato con el almacén de datos, y es incorrecto.

Consistencia causal

El modelo de **consistencia causal** (Hutto y Ahamad, 1990) representa una debilidad de la consistencia secuencial, ya que diferencia entre eventos que potencialmente están relacionados por la causalidad y los que no lo están. En el capítulo anterior, cuando explicamos los registros de tiempo vectoriales, ya tratamos con la causalidad. Si el evento *b* es causado o influenciado por un evento previo *a*, la causalidad requiere que todos los demás eventos vean primero a *a*, y después a *b*.

Considere una simple interacción mediante una base de datos distribuida compartida. Suponga que el proceso P_1 escribe un elemento de datos *x*. Después P_2 lee a *x* y escribe *y*. Aquí, la lectura de *x* y la escritura de *y* están relacionados potencialmente por la causalidad, ya que el cálculo de *y* pudo haber dependido del valor de *x* cuando P_2 lo leyó (es decir, el valor escrito por P_1).

Por otra parte, si dos procesos escriben espontánea y simultáneamente dos diferentes elementos de datos, éstos no están causalmente relacionados. Se dice que las operaciones que no están causalmente relacionadas son **concurrentes**.

Para que a un almacén de datos se le considere causalmente consistente, es necesario que obedezca la siguiente condición:

Escrituras que potencialmente están relacionadas por la causalidad, deben ser vistas por todos los procesos en el mismo orden. Las escrituras concurrentes pueden verse en un orden diferente en diferentes máquinas.

Como un ejemplo de consistencia causal, consideremos la figura 7-8. Aquí tenemos una secuencia de eventos que está permitida con un almacén causalmente consistente, pero que está prohibida con un almacén secuencialmente consistente o con un almacén estrictamente consistente. El punto a destacar es que las escrituras $W_2(x)b$ y $W_1(x)c$ son concurrentes, por ello no se requiere que todos los procesos las vean en el mismo orden.

P1: W(x)a			W(x)c		
P2:	R(x)a	W(x)b			
P3:	R(x)a			R(x)c	R(x)b
P4:	R(x)a			R(x)b	R(x)c

Figura 7-8. Esta secuencia está permitida con un almacén causalmente consistente, pero no con un almacén secuencialmente consistente.

Ahora consideremos un segundo ejemplo. En la figura 7-9(a) tenemos a $W_2(x)b$ potencialmente dependiente de $W_1(x)a$, ya que b puede ser el resultado de un cálculo que involucra al valor leído por $R_2(x)a$. Las dos escrituras están causalmente relacionadas, por lo que todos los procesos deben verlas en el mismo orden. Por tanto, la figura 7-9(a) es incorrecta. Por otra parte, en la figura 7-9(b) la lectura ha sido eliminada, así que $W_1(x)a$ y $W_2(x)b$ ahora son escrituras concurrentes. Un almacén causalmente consistente no requiere de escrituras concurrentes globalmente ordenadas, por lo que la figura 7-9(b) es correcta. Observe que la figura 7-9(b) refleja una situación que no sería aceptable para un almacén secuencialmente consistente.

P1: W(x)a				
P2:	R(x)a	W(x)b		
P3:			R(x)b	R(x)a
P4:			R(x)a	R(x)b

(a)

P1: W(x)a			
P2:	W(x)b		
P3:		R(x)b	R(x)a
P4:		R(x)a	R(x)b

(b)

Figura 7-9. (a) Violación a un almacén causalmente consistente. (b) Secuencia de eventos correcta en un almacén causalmente consistente.

Implementar la consistencia causal requiere dar seguimiento a cuáles procesos han visto cuáles escrituras. En efecto, esto significa que debe construirse y mantenerse una gráfica de la dependencia

de cuál operación depende de qué otras operaciones. Una forma de hacerlo es mediante un registro de tiempo vectorial, como explicamos en el capítulo anterior. Más adelante en este capítulo retomaremos el uso de registros de tiempo vectoriales para capturar la causalidad.

Operaciones de agrupamiento

La consistencia secuencial y la causal están definidas al nivel de operaciones de lectura y escritura. Este nivel de granularidad se debe a razones históricas: estos modelos se desarrollaron inicialmente para sistemas de multiprocesador de memoria compartida, y en realidad se implementaron al nivel de hardware.

La granularidad fina de estos modelos de consistencia no coincide en muchos casos con la granularidad provista por las aplicaciones. Lo que vemos ahí es que la concurrencia entre programas que comparten datos generalmente se mantiene bajo control a través de mecanismos de sincronización para exclusión mutua y transacciones. En efecto, ocurre que a nivel de programa, las operaciones de lectura y escritura se colocan entre corchetes mediante el par de operaciones ENTER_CS y LEAVE_CS, donde "CS" significa sección crítica. Como explicamos en el capítulo 6, la sincronización entre procesos se lleva a cabo mediante estas dos operaciones. En términos de nuestro almacén de datos distribuido, esto significa que un proceso que ha ejecutado exitosamente ENTER_CS estará seguro de que los datos de su almacén local están actualizados. En ese punto, el proceso puede ejecutar con seguridad una serie de operaciones de lectura y escritura en ese almacén, y posteriormente terminar mediante la llamada a LEAVE_CS.

En esencia, ocurre que dentro de un programa, los datos manejados mediante una serie de operaciones de lectura y escritura están protegidos contra accesos concurrentes que ocasionarían ver algo diferente al resultado de la ejecución de la serie como un todo. Puesto de otro modo, los corchetes convierten la serie de operaciones de lectura y escritura en una unidad ejecutada atómicamente, y de esta manera aumentan el nivel de granularidad.

Para llegar a este punto necesitamos una semántica precisa de las operaciones ENTER_CS y LEAVE_CS. Esta semántica puede formularse en términos de **variables de sincronización** compartidas. Existen diferentes formas de utilizar estas variables. Veremos un método general en el que cada variable tiene algún dato asociado, el cual podría agregarse al conjunto total de datos compartidos. Nosotros adoptamos la convención de que cuando un proceso entra en su sección crítica, debe *adquirir* las variables importantes de sincronización; de igual manera, cuando abandona la sección crítica, debe *liberar* estas variables. Observe que los datos incluidos en la sección crítica del proceso pueden asociarse con diferentes variables de sincronización.

Cada variable de sincronización tiene un propietario actual, a saber, el último proceso que la adquirió. El propietario puede entrar y salir repetidamente de secciones críticas sin tener que enviar mensaje alguno en la red. Un proceso que actualmente no posee una variable de sincronización, pero que quiere adquirirla, tiene que enviar un mensaje al propietario actual solicitándole su propiedad y los valores actuales de los datos asociados con esa variable de sincronización. También es posible que varios procesos posean simultáneamente una variable de sincronización de manera no exclusiva, lo cual significa que pueden leer, pero no escribir, los datos asociados.

Ahora necesitamos que se cumplan los siguientes criterios (Bershad y cols., 1993):

1. *El acceso para adquirir una variable de sincronización con respecto a un proceso no está permitido sino hasta que se realizan todas las actualizaciones de los datos compartidos con respecto a ese proceso.*

2. *Antes de que a un proceso se le permita un modo exclusivo de acceso a una variable de sincronización, ningún otro proceso puede tener a la variable de sincronización, ni siquiera en modo no exclusivo.*

3. *Después de que se ha realizado un acceso en modo exclusivo hacia una variable de sincronización, ningún otro acceso de modo no exclusivo de otro proceso hacia esa variable de sincronización puede realizarse, sino hasta que se haya realizado con respecto al propietario de esa variable.*

La primera condición establece que cuando un proceso hace una adquisición, ésta no puede completarse (es decir, devolver el control a la siguiente instrucción) sino hasta que todos los datos compartidos guardados se han actualizado. En otras palabras, en una adquisición, todos los cambios remotos a los datos guardados deben hacerse visibles.

La segunda condición establece que antes de actualizar un elemento de datos compartido, un proceso debe entrar a su sección crítica en modo exclusivo para garantizar que ningún otro proceso está intentando actualizar los datos compartidos al mismo tiempo.

La tercera condición establece que si un proceso quiere entrar a una región crítica de modo no exclusivo, primero debe verificar con el propietario de la variable de sincronización que guarda la región crítica para buscar las copias más recientes de los datos guardados compartidos.

La figura 7-10 muestra un ejemplo de lo que se conoce como **consistencia de entrada**. En lugar de operar sobre todos los datos compartidos, en este ejemplo asociamos candados con cada elemento de datos. En este caso, P_1 hace una adquisición para *x*, cambia *x* una vez, después de lo cual también hace una adquisición para *y*. El proceso P_2 hace una adquisición para *x* pero no para *y*, por lo que leerá el valor *a* para *x*, pero puede leer *NIL* para *y*. Debido a que el proceso P_3 realiza primero una adquisición para *y*, leerá el valor *b* cuando *y* sea liberado por P_1.

P1:	Acq(Lx)	W(x)a	Acq(Ly)	W(y)b	Rel(Lx)	Rel(Ly)			
P2:						Acq(Lx)	R(x)a		R(y) NIL
P3:							Acq(Ly)	R(y)b	

Figura 7-10. Secuencia de eventos válida para consistencia de entrada.

Uno de los problemas de programar con consistencia de entrada es asociar adecuadamente los datos con variables de sincronización. Un método directo es indicarle explícitamente al middleware a cuáles datos se va a acceder, como generalmente se hace al declarar cuáles tablas de la base de datos serán afectadas por una transacción. En un método basado en objetos, podríamos asociar

implícitamente una variable de sincronización única con cada objeto declarado, con lo que pondríamos efectivamente en serie todas las invocaciones a tales objetos.

Consistencia *versus* coherencia

En este punto, resulta útil aclarar la diferencia entre dos conceptos muy relacionados. Los modelos que hemos explicado hasta el momento tratan con el hecho de que varios procesos ejecutan operaciones de lectura y escritura sobre un conjunto de elementos de datos. Un **modelo de consistencia** describe lo que se puede esperar con respecto a ese conjunto cuando varios procesos operan concurrentemente sobre esos datos. Entonces, se dice que el conjunto es consistente si se apega a las reglas descritas por el modelo.

Mientras que la consistencia trata con un conjunto de elementos de datos, los **modelos de coherencia** describen lo que puede esperarse de un solo elemento de datos (Cantin y cols., 2005). En este caso, suponemos que un elemento de datos se replica en varios lugares; se dice que es coherente cuando las diversas copias se apegan a las reglas definidas por su modelo de coherencia asociado. Un popular modelo es el de consistencia secuencial, pero aplicado ahora a un solo elemento de datos. En efecto, esto significa que en el caso de escrituras concurrentes, todos los procesos verán en algún momento que tiene lugar el mismo orden de actualizaciones.

7.3 MODELOS DE CONSISTENCIA CENTRADA EN EL CLIENTE

Los modelos de consistencia que describimos en la sección anterior ayudan a proporcionar una vista consistente a lo largo del sistema de un almacén de datos. Una suposición importante es que procesos concurrentes pueden actualizar simultáneamente el almacén de datos, y que es necesario proporcionar consistencia ante dicha concurrencia. Por ejemplo, en el caso de la consistencia de entrada basada en objetos, el almacén de datos garantiza que cuando un objeto es llamado, el proceso que llama es provisto con una copia del objeto que refleja todos los cambios hechos al objeto hasta el momento, probablemente por otros procesos. Durante la llamada, también se garantiza que ningún otro proceso puede interferir; es decir, se le proporciona acceso mutuo exclusivo al proceso que llama.

Ser capaz de manejar operaciones concurrentes sobre datos compartidos, mientras se mantiene la consistencia secuencial, es fundamental para los sistemas distribuidos. Por razones de rendimiento, la consistencia secuencial probablemente puede garantizarse sólo cuando los procesos utilizan mecanismos de sincronización tales como transacciones o candados.

En esta sección, veremos una clase especial de almacenes de datos distribuidos. Los almacenes de datos que consideramos se caracterizan por la falta de actualizaciones simultáneas, o cuando tales actualizaciones ocurren, pueden resolverse fácilmente. La mayoría de las operaciones involucran la lectura de datos. Estos almacenes de datos ofrecen un modelo de consistencia muy débil, llamado de consistencia momentánea. Al introducir modelos especiales de consistencia centrada en el cliente, se vuelve evidente que es posible ocultar muchas inconsistencias de una manera relativamente barata.

7.3.1 Consistencia momentánea

Hasta qué punto los procesos en realidad operan de manera concurrente, y hasta qué punto la consistencia necesita garantizarse, puede variar. Hay muchos ejemplos, en los que la concurrencia aparece sólo en forma restrictiva. Por ejemplo, en muchos sistemas de bases de datos, la mayoría de los procesos difícilmente realizan alguna vez operaciones de actualización; en su mayoría leen datos de la base de datos. Sólo uno, o muy pocos procesos realizan operaciones de actualización. Entonces, la pregunta es qué tan rápido deben estar disponibles las actualizaciones para los procesos de sólo lectura.

Como otro ejemplo, considere un sistema mundial de asignación de nombres como el DNS. El espacio de nombre DNS se divide en dominios, donde cada dominio es asignado a una autoridad de asignación que actúa como propietaria de ese dominio. Sólo a esa autoridad se le permite actualizar su parte del espacio de nombre. En consecuencia, conflictos entre dos operaciones que desean realizar una actualización sobre el mismo dato (es decir, **conflictos de escritura-escritura**) nunca ocurren. La única situación que necesita manejarse son los **conflictos de lectura-escritura**, en la que un proceso desea actualizar un elemento de datos mientras otro intenta, concurrentemente, leer ese elemento. Como resultado, con frecuencia es aceptable propagar una actualización de modo lento, lo que significa que un proceso de lectura verá una actualización sólo cierto tiempo después de que la actualización ocurrió.

Otro ejemplo es la World Wide Web. En casi todos los casos, las páginas web son actualizadas por una sola autoridad, como un webmaster o el propietario real de la página. Normalmente no hay conflictos de escritura-escritura por resolver. Por otra parte, para mejorar la eficiencia, los navegadores y proxies web con frecuencia se configuran para mantener páginas buscadas en un caché local y devolverlas en la siguiente petición.

Un aspecto importante de ambos tipos de caché web es que ambos pueden devolver páginas web no actualizadas. En otras palabras, la página cacheada que es devuelta al cliente solicitante es una versión antigua, comparada con la disponible en el servidor real web. Como resultado, muchos usuarios encuentran aceptable (hasta cierto punto) esta inconsistencia.

Estos ejemplos pueden considerarse como casos de bases de datos replicadas y distribuidas (de gran escala) que toleran un relativamente alto grado de inconsistencia. Tienen en común que, si no ocurren actualizaciones durante mucho tiempo, todas las réplicas gradualmente se volverán inconsistentes. Esta forma de consistencia se conoce como **consistencia momentánea**.

Los almacenes de datos que son momentáneamente consistentes tienen la propiedad de que, en ausencia de actualizaciones, todas las réplicas convergen en copias idénticas unas de otras. En esencia, la consistencia momentánea sólo requiere la garantía de que las actualizaciones se propaguen a todas las réplicas. Los conflictos de escritura-escritura con frecuencia son fáciles de resolver cuando se asume que sólo un pequeño grupo de procesos puede realizar actualizaciones. La implementación de la consistencia momentánea es, por tanto, barata.

Los almacenes de datos consistentes momentáneamente funcionan bien siempre y cuando los clientes siempre accedan a la misma réplica. Sin embargo, los problemas surgen cuando en un periodo corto se accede a réplicas diferentes. Esto se ilustra mejor si se considera a un usuario móvil accediendo a una base de datos distribuida, como se muestra en la figura 7-11.

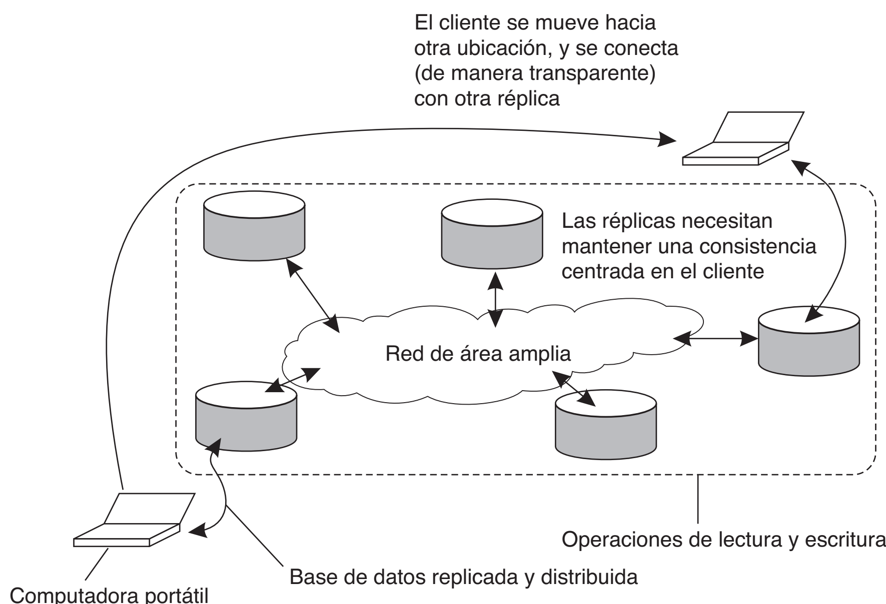


Figura 7-11. El principio referente a un usuario móvil que accede a diferentes réplicas de una base de datos distribuida.

El usuario móvil accede a la base de datos conectándose de manera transparente a una de las réplicas. En otras palabras, la aplicación que está en ejecución en la computadora portátil del cliente no se percata de en qué réplica opera realmente. Suponga que el usuario realiza varias operaciones de actualización y después se desconecta de nuevo. Más tarde accede otra vez a la base de datos, probablemente después de moverse a una ubicación diferente o de utilizar un dispositivo de acceso distinto. En ese punto, el usuario puede estar conectado a una réplica diferente a la anterior, como ilustra la figura 7-11. Sin embargo, si las actualizaciones realizadas antes aún no se han propagado, el usuario notará un comportamiento inconsistente. En particular, esperaríamos ver todos los cambios realizados antes, pero en vez de eso parece como si nada en absoluto hubiese pasado.

Este ejemplo es típico de almacenes de datos momentáneamente consistentes, y se origina por que los usuarios en ocasiones pueden operar sobre réplicas diferentes. El problema puede aligerarse introduciendo la **consistencia centrada en el cliente**. En esencia, la consistencia centrada en el cliente proporciona garantías *para un solo cliente* con respecto a la consistencia de acceso al almacén de datos de ese cliente. No se proporciona garantía alguna con respecto a accesos concurrentes de diferentes clientes.

Los modelos de consistencia centrada en el cliente se originaron a partir del trabajo sobre Bayou [por ejemplo, vea Terry y cols. (1994), y Terry y cols. (1998)]. Bayou es un sistema de bases de datos desarrollado para computación móvil, en donde se asume que la conectividad de la red no es confiable y que está sujeta a diversos problemas de rendimiento. Las redes inalámbricas y las que abarcan grandes áreas, como internet, caen en esta categoría.

Bayou esencialmente distingue cuatro tipos diferentes de modelos de consistencia. Para explicar estos modelos, consideremos nuevamente un almacén de datos que está físicamente distribuido a través de varias máquinas. Cuando un proceso accede al almacén de datos, generalmente se conecta a la copia local (o más cercana) disponible, aunque, en principio, cualquier copia estaría bien. Todas las operaciones de lectura y escritura se realizan en esa copia local. Las actualizaciones se propagan, en algún momento, a las otras copias. Para simplificar las cosas, suponemos que los elementos de datos tienen un propietario asociado, el cual es el único proceso que tiene permitido modificar ese elemento. De esta manera evitamos los conflictos de escritura-escritura.

Los modelos de consistencia centrada en el cliente se describen mediante la siguiente notación. Sea $x_i[t]$ quien denote la versión del elemento de datos x en la copia local L_i al tiempo t. La versión $x_i[t]$ es el resultado de una serie de operaciones de escritura en L_i que ocurre desde la inicialización. Denotamos este conjunto como $WS(x_i[t])$. Si las operaciones en $WS(x_i[t_1])$ también se realizaron en la copia local L_j en un tiempo posterior t_2, escribimos $WS(x_i[t_1];x_j[t_2])$. Si el ordenamiento de las operaciones o la sincronización es algo que queda claro por el contexto, omitiremos el índice del tiempo.

7.3.2 Lecturas monotónicas

El primer modelo de consistencia centrada en el cliente es el de lecturas monotónicas. Se dice que un almacén de datos proporciona **consistencia de lectura monotónica** si se cumple la siguiente condición:

> *Si un proceso lee el valor de un elemento de datos* x*, cualquier operación de lectura sucesiva sobre* x *que haga ese proceso devolverá siempre el mismo valor o un valor más reciente.*

En otras palabras, la consistencia de lectura monotónica garantiza que si un proceso ha visto un valor de x al tiempo t, nunca verá una versión más vieja de x en un tiempo posterior.

Como un ejemplo de en dónde son útiles las lecturas monotónicas, considere una base de datos distribuida de correo electrónico. En tal base de datos, el buzón electrónico de cada usuario puede distribuirse y replicarse a través de varias máquinas. El correo puede insertarse en un buzón electrónico en cualquier ubicación. Sin embargo, las actualizaciones se propagan lentamente (es decir, a demanda). Sólo cuando una copia necesita ciertos datos por consistencia, esos datos se propagan a esa copia. Suponga que un usuario lee su correo en San Francisco. Suponga que la sola lectura de correos no afecta el buzón, es decir, los mensajes no se eliminan, se almacenan en subdirectorios, o incluso se marcan como leídos, etc. Cuando el usuario vuela posteriormente hacia Nueva York y abre nuevamente su buzón electrónico, la consistencia de lectura monotónica garantiza que los mensajes que se encontraban en dicho buzón en San Francisco también estarán en el buzón cuando se abra en Nueva York.

Si utilizamos una notación similar a la que usamos para los modelos de consistencia centrada en los datos, la consistencia de lectura monotónica puede representarse gráficamente como ilustra la figura 7-12. A lo largo del eje vertical, aparecen dos diferentes copias locales del almacén de datos, L_1 y L_2. El tiempo aparece a lo largo del eje horizontal, como antes. En todos los casos, nos

interesamos en las operaciones realizadas por un solo proceso, *P*. Estas operaciones específicas aparecen en negritas y están conectadas por una línea punteada que representa el orden en que las realiza *P*.

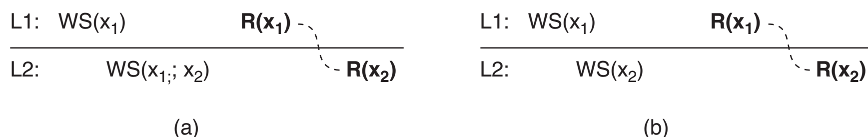


Figura 7-12. Operaciones de lectura realizadas por un solo proceso, *P*, en dos diferentes copias del mismo almacén de datos. (a) Almacén de datos con consistencia de lectura monotónica. (b) Almacén de datos que no proporciona lecturas monotónicas.

En la figura 7-12(a), el proceso *P* realiza primero una operación de lectura sobre *x* en L_1, y devuelve el valor de x_1 (en ese tiempo). Este valor resulta de las operaciones de escritura sobre $WS(x_1)$ realizadas en L_1. Después, *P* realiza una operación de lectura sobre *x* en L_2, mostrada como $R(x_2)$. Para garantizar la consistencia de lectura monotónica, todas las operaciones sobre $WS(x_1)$ deben haberse propagado a L_2 antes de que ocurra la segunda operación de lectura. En otras palabras, necesitamos saber con certeza que $WS(x_1)$ es parte de $WS(x_2)$, lo cual se expresa como $WS(x_1;x_2)$.

Por contraste, la figura 7-12(b) muestra una situación en la que no se garantiza la consistencia de lectura monotónica. Después de que el proceso *P* lee x_1 en L_1, realiza la operación $R(x_2)$ en L_2. Sin embargo, solamente las operaciones de escritura sobre $WS(x_2)$ se han realizado en L_2. No hay garantía de que este conjunto incluya también todas las operaciones contenidas en $WS(x_1)$.

7.3.3 Escrituras monotónicas

En muchas situaciones, es importante que las operaciones de escritura se propaguen en el orden correcto hacia todas las copias del almacén de datos. Esta propiedad se expresa en consistencia de escritura monotónica. En un almacén con **consistencia de escritura monotónica**, se cumple la siguiente condición:

> *Una operación de escritura hecha por un proceso sobre un elemento* x *se completa antes que cualquier otra operación sucesiva de escritura sobre* x *realizada por el mismo proceso.*

Así, completar una operación de escritura significa que la copia sobre la cual se realiza una operación sucesiva refleja el efecto de una operación de escritura previa realizada por el mismo proceso, independientemente de dónde se inició esa operación. En otras palabras, una operación de escritura sobre una copia del elemento *x* se realiza sólo si esa copia se ha actualizado mediante cualquier operación de escritura previa, la cual pudo haber ocurrido en otras copias de *x*. Si es necesario, la nueva escritura debe esperar a que terminen otras escrituras anteriores.

Observe que la consistencia de escritura monotónica se parece a la consistencia PEPS centrada en los datos. Lo básico de la consistencia PEPS es que las operaciones de escritura del mismo proceso se realizan en el orden correcto en cualquier parte. Esta restricción de ordenamiento también aplica para escrituras monotónicas, con la excepción de que ahora sólo consideramos consistencia para un solo proceso en lugar de hacerlo para una colección de procesos concurrentes.

Actualizar una copia de *x* no es necesario cuando cada operación de escritura sobreescribe por completo el valor actual de *x*. Sin embargo, las operaciones de escritura con frecuencia se realizan sólo en una parte del estado de un elemento de datos. Por ejemplo, consideremos una biblioteca de software. En muchos casos, la actualización de una biblioteca de este tipo se hace reemplazando una o más funciones, lo que deriva en una siguiente versión. Con la consistencia de escritura monotónica se proporcionan garantías de que si una actualización se realiza sobre una copia de la biblioteca, todas las actualizaciones anteriores se realizarán primero. La biblioteca resultante se volverá entonces la versión más reciente, e incluirá todas las actualizaciones que han dado pie a versiones anteriores de la biblioteca.

La figura 7-13 muestra la consistencia de escritura monotónica. En el inciso (a) de dicha figura, el proceso *P* realiza una operación de escritura sobre *x* en la copia local L_1, y se presenta como la operación $W(x_1)$. Después, *P* realiza otra operación de escritura sobre *x*, pero esta vez en L_2, y se muestra como $W(x_2)$. Para garantizar la consistencia de escritura monotónica, es necesario que las operaciones de escritura anteriores realizadas en L_1 se hayan propagado a L_2. Esto explica la operación $W(x_1)$ en L_2, y por qué ocurre antes que $W(x_2)$.

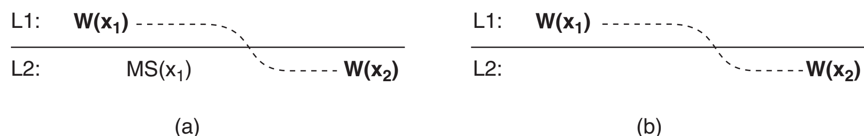


Figura 7-13. Operaciones de escritura realizadas por un solo proceso *P* en dos copias locales diferentes del mismo almacén de datos. (a) Almacén de datos con consistencia de escritura monotónica. (b) Almacén de datos que no proporciona consistencia de escritura monotónica.

Por contraste, la figura 7-13(b) muestra una situación en la que no se garantiza la consistencia de escritura monotónica. Comparada con la figura 7-13(a), lo que falta es la propagación de $W(x_1)$ a la copia L_2. En otras palabras, no es posible garantizar que la copia de *x*, en la que se está realizando la segunda escritura, tenga el mismo o el más reciente valor en el tiempo $W(x_1)$ completado en L_1.

Observe que, por definición de la consistencia de escritura monotónica, las operaciones de escritura del mismo proceso se realizan en el mismo orden en que iniciaron. Una forma en cierto modo más débil de las escrituras monotónicas es una en la que los efectos de una operación de escritura sólo se ven si todas las escrituras anteriores se han llevado a cabo, aunque quizá no en el orden en que se originaron inicialmente. Esta consistencia es aplicable a aquellos casos en que las operaciones de escritura son conmutativas, de modo que el ordenamiento en realidad no es necesario. Pueden encontrarse los detalles en Terry y colaboradores (1994).

7.3.4 Lea sus escrituras

Un modelo de consistencia centrado en el cliente que está muy relacionado con lecturas monotónicas es como sigue. Se dice que un almacén de datos proporciona **consistencia lea sus escrituras** si cumple la siguiente condición:

> *El efecto de una operación de escritura hecha por un proceso sobre un elemento de datos* x *siempre será visto por una operación de lectura sucesiva sobre* x *hecha por el mismo proceso.*

En otras palabras, una operación de escritura siempre se completa antes de una operación de lectura sucesiva del mismo proceso, independientemente del lugar donde ocurra la operación de lectura.

En ocasiones, la falta de consistencia lea sus escrituras se experimenta cuando se actualizan documentos web, y posteriormente se ven los efectos. Las operaciones de actualización frecuentemente ocurren por medio de un editor estándar o un procesador de palabras, los cuales guardan la nueva versión en un sistema de archivos que es compartido por el servidor web. El navegador web del usuario accede a ese mismo archivo, probablemente después de solicitárselo al servidor web local. Sin embargo, una vez que el archivo ha sido traído, ya sea el servidor o el navegador a menudo cachean una copia local para accesos posteriores. En consecuencia, cuando la página web se actualiza, si el navegador o el servidor devuelven la copia cacheada en lugar del archivo original, el usuario no verá los efectos. La consistencia lea sus escrituras puede garantizar que si el editor y el navegador se integran en un solo programa, el caché se invalida cuando la página es actualizada, por lo que el archivo actualizado es traído y desplegado.

Efectos similares se presentan al actualizar contraseñas. Por ejemplo, para ingresar a una biblioteca digital en la web, con frecuencia es necesario tener una cuenta y su contraseña correspondiente. Sin embargo, para que el cambio de una contraseña surta efecto puede necesitarse tiempo, con el resultado de que la biblioteca sería inaccesible para el usuario durante algunos minutos. El retraso puede ser causado porque se utiliza un servidor por separado para manejar contraseñas, y puede llevarse algún tiempo la propagación subsiguiente de contraseñas (encriptadas) hacia los diferentes servidores que constituyen la biblioteca.

La figura 7-14(a) muestra un almacén de datos que proporciona consistencia lea sus escrituras. Observe que la figura 7-14(a) es muy similar a la figura 7-12(a), con la excepción de que ahora la consistencia es determinada por la última operación de escritura del proceso *P*, en lugar de su última lectura.

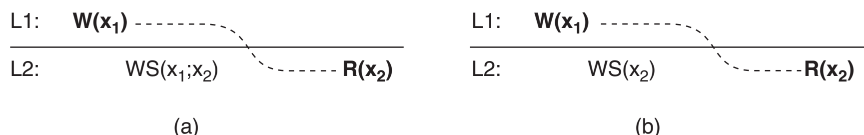


Figura 7-14. (a) Almacén de datos que proporciona consistencia lea sus escrituras. (b) Almacén de datos que no la proporciona.

En la figura 7-14(a), el proceso *P* realizó una operación de escritura $W(x_1)$ y después una operación de lectura en una copia local distinta. La consistencia lea sus escrituras garantiza que los

efectos de la operación de escritura puedan ser vistos por la sucesiva operación de lectura. Esto se expresa mediante $WS(x_1;x_2)$, lo cual establece que $W(x_1)$ es parte de $WS(x_2)$. Por contraste, en la figura 7-14(b) $W(x_1)$ ha sido excluida de $WS(x_2)$, esto significa que los efectos de la operación de escritura anterior del proceso *P* aún no se han propagado a L_2.

7.3.5 Las escrituras siguen a las lecturas

El último modelo de consistencia centrada en el cliente es uno en el que las actualizaciones se propagan como resultado de operaciones de lectura previas. Se dice que un almacén de datos proporciona consistencia **las escrituras siguen a las lecturas** si cumple con lo siguiente:

> *Se garantiza que la operación de escritura de un proceso sobre un elemento de datos x que sigue a una operación de lectura previa sobre* x *efectuada por el mismo proceso ocurrirá en el mismo o en el más reciente valor de* x *que se leyó.*

En otras palabras, cualquier operación sucesiva de un proceso sobre un elemento de datos *x* se realizará sobre una copia de *x* que está actualizada con el valor más recientemente leído por ese proceso.

La consistencia las escrituras siguen a las lecturas puede utilizarse para garantizar que los usuarios de una red de un grupo de noticias vean el anuncio de una respuesta a un artículo sólo después de que han visto el artículo original (Terry y cols., 1994). Para entender el problema, suponga que un usuario lee primero el artículo *A*; después, reacciona y publica la respuesta *B*. Al requerir consistencia las escrituras siguen a las lecturas, *B* será escrita en cualquier copia del grupo de noticias sólo después de que *A* también sea escrita. Observe que los usuarios que sólo leen los artículos no necesariamente requieren un modelo específico de consistencia centrada en el cliente. La consistencia las escrituras siguen a las lecturas garantiza que las reacciones a los artículos se almacenen en una copia local sólo si el original también se almacena ahí.

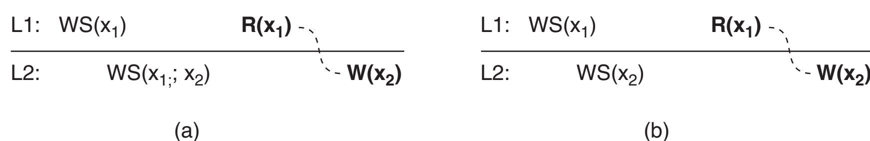


Figura 7-15. (a) Almacén de datos con consistencia las escrituras siguen a las lecturas. (b) Almacén de datos que no proporciona consistencia las escrituras siguen a las lecturas.

Este modelo de consistencia aparece en la figura 7-15. En el inciso (a) de esta figura, un proceso lee a *x* en una copia local L_1. Las operaciones de escritura que ocasionaron el valor apenas leído, también aparecen en el conjunto escrito en L_2, donde el mismo proceso realiza después una operación de escritura. (Observe que otros procesos incluidos en L_2 también ven aquellas operaciones de escritura.) En cambio, no se garantiza que la operación realizada en L_2, como ilustra la figura 7-15(b), sea realizada en una copia consistente con la recién leída en L_1.

Más adelante en este capítulo retomaremos los modelos de consistencia centrados en el cliente, cuando expliquemos las implementaciones.

7.4 ADMINISTRACIÓN DE RÉPLICAS

Un punto clave para cualquier sistema distribuido que soporta la replicación es decidir dónde, cuándo, y por quién deben ubicarse las réplicas, y posteriormente cuáles mecanismos utilizar para mantener consistentes a dichas réplicas. El problema de ubicación, por sí mismo, debe dividirse en dos subproblemas: el de la ubicación de *servidores de réplicas*, y el de *ubicación de contenido*. La diferencia es sutil pero importante, y con frecuencia las dos cuestiones no están claramente separadas. La ubicación del servidor de réplicas tiene que ver con encontrar los mejores lugares para colocar un servidor que pueda hospedar (parte de) un almacén de datos. La ubicación de contenido se relaciona con encontrar a los mejores servidores para colocar el contenido. Observe que, generalmente, esto significa que buscamos la ubicación óptima de un solo elemento de datos. Desde luego, antes de que la ubicación de contenido pueda ocurrir, primero tenemos que ubicar los servidores. A continuación, daremos un vistazo a estos dos problemas de ubicación, y continuaremos con una explicación sobre los mecanismos básicos para administrar el contenido replicado.

7.4.1 Ubicación del servidor de réplicas

La ubicación de servidores de réplicas no es un problema estudiado intensivamente, por la simple razón de que a menudo se trata de un asunto más administrativo y comercial que un problema de optimización. No obstante, el análisis de las propiedades del cliente y de la red es útil para tomar decisiones informadas.

Existen varias formas de calcular la mejor ubicación de servidores de réplicas, pero todas se reducen a un problema de optimización en el que se necesita seleccionar la mejor K de entre N ubicaciones ($K < N$). Se sabe que estos problemas son de cómputo complejo, y que sólo pueden resolverse mediante la heurística. Qiu y colaboradores (2001) toman la distancia entre clientes y ubicaciones desde un punto de partida. La distancia puede medirse en términos de latencia o ancho de banda. Su solución selecciona un servidor a un tiempo tal que la distancia promedio entre ese servidor y sus clientes sea mínima dado que k servidores ya han sido ubicados (ello significa que hay $N - k$ ubicaciones descartadas).

Como alternativa, Radoslavov y colaboradores (2001) proponen ignorar la posición de los clientes, y sólo toman la topología de internet como si estuviera formada por sistemas autónomos. Un **sistema autónomo** (**AS**, por sus siglas en inglés) puede considerarse como una red en la que todos los nodos ejecutan el mismo protocolo de enrutamiento, y que es administrada por una sola organización. Para enero del 2006, ya había más de 20 000 sistemas autónomos. Radoslavov y colaboradores consideran primero el AS más grande y colocan un servidor en el ruteador que tenga el mayor número de interfaces de red (es decir, vínculos). Después este algoritmo se repite con el segundo AS más grande, y así sucesivamente.

Como resultado, la ubicación de un servidor sin el conocimiento del cliente logra resultados similares a los obtenidos con conocimiento del cliente, bajo la suposición de que los clientes están distribuidos uniformemente a través de internet (según la topología existente). Hasta qué punto sea cierta esta suposición, no queda claro; aún no ha sido bien estudiada.

Un problema con estos algoritmos es que son de cómputo caro. Por ejemplo, los dos algoritmos previos tienen una complejidad mayor a O(N^2), donde N es el número de ubicaciones por inspeccionar. En la práctica, esto significa que incluso para algunas miles de ubicaciones un cálculo puede necesitar ejecutarse por decenas de minutos. Esto puede resultar inaceptable, sobretodo cuando se presentan **flash crowds** (un súbito estallido de peticiones para un sitio específico, lo cual ocurre con regularidad en internet). En ese caso, es básico determinar rápidamente en dónde se necesitan los servidores de réplicas, después de lo cual puede seleccionarse un servidor para la ubicación de contenido.

Szymaniak y colaboradores (2006) desarrollaron un método mediante el cual puede identificarse rápidamente una región para ubicar réplicas. Una región se identifica para ser una colección de nodos que accede al mismo contenido, pero para la cual la latencia internodal es baja. El objetivo del algoritmo es seleccionar primero las regiones con más demanda, es decir, aquellas con más nodos, y después dejar que uno de los nodos de esa región actúe como servidor de réplicas.

Con este fin, se asume que los nodos están posicionados en un espacio geométrico m dimensional, como explicamos en el capítulo anterior. La idea básica es identificar los K cluster más grandes y asignar un nodo de cada cluster para que hospede el contenido replicado. Para identificar estos cluster, todo el espacio se divide en celdas. Las K celdas más densas se eligen entonces para colocar un servidor de réplicas. Una celda no es más que un hipercubo m dimensional. Para un espacio bidimensional, esto corresponde a un rectángulo.

Desde luego, el tamaño de la celda es importante, como se muestra en la figura 7-16. Si las celdas se eligen demasiado grandes, entonces varios cluster de nodos pueden encontrarse en la misma celda. En ese caso, se elegirían muy pocos servidores de réplicas para esos cluster. Por otra parte, elegir celdas pequeñas puede ocasionar que un solo cluster se propague a través de varias celdas, lo cual ocasionaría la elección de demasiados servidores de réplicas.

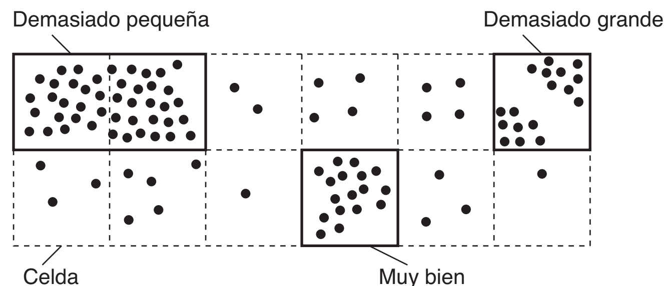


Figura 7-16. Elección del tamaño adecuado de una celda para ubicar un servidor.

Como resultado, un tamaño adecuado de celda puede calcularse como una simple función de la distancia promedio existente entre dos nodos y el número de réplicas requeridas. Con dicho tamaño de celda puede mostrarse que el algoritmo se comporta tan bien como el casi óptimo descrito por Qiu y colaboradores (2001), pero con un grado de complejidad mucho menor: O($N \times$xmáx{$log(N)$,K}). Para ilustrar lo que significa este resultado: los experimentos muestran

que calcular las 20 mejores ubicaciones de réplicas para una colección de 64 000 nodos es aproximadamente 50000 veces más rápido. En consecuencia, la ubicación de servidores de réplicas puede hacerse ahora en tiempo real.

7.4.2 Ubicación y replicación de contenido

Ahora dejemos la ubicación de servidores y concentrémonos en la ubicación de contenido. Cuando se trata de replicación y ubicación de contenido, es posible diferenciar tres tipos de réplicas lógicamente organizadas, tal como ilustra la figura 7-17.

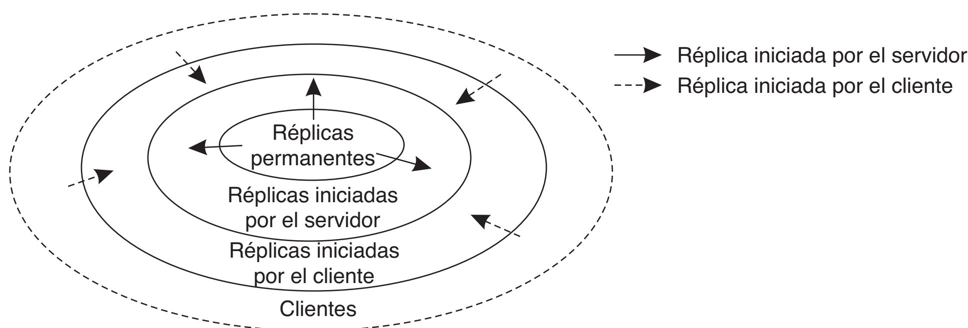


Figura 7-17. Organización lógica de distintos tipos de copias de un almacén de datos dentro de tres anillos concéntricos.

Réplicas permanentes

Las réplicas permanentes pueden considerarse como el conjunto inicial de réplicas que constituyen un almacén de datos distribuido. En muchos casos, el número de réplicas permanentes es pequeño. Por ejemplo, consideremos un sitio web. La distribución de un sitio web, por lo general, viene en una de dos formas. La primera clase de distribución es aquella en la cual los archivos que constituyen un sitio se replican a través de un número limitado de servidores en una sola ubicación. Siempre que llega una petición, es reenviada a uno de los servidores, por ejemplo, utilizando una estrategia *round-robin* (todos contra todos).

La segunda forma de sitios web distribuidos es la que se conoce como **espejear**. En este caso, un sitio web es copiado en un número limitado de servidores, llamados **sitios espejo**, los cuales están geográficamente distribuidos a través de internet. En la mayoría de los casos, los clientes simplemente eligen uno de los diferentes sitios espejo desde una lista que se les ofrece. Los sitios web reflejados tienen en común sitios web basados en cluster que sólo tienen unas cuantas réplicas, las cuales son configuradas más o menos estáticamente.

Organizaciones estáticas similares también se presentan con bases de datos distribuidas (Oszu y Valduriez, 1999). De nuevo, la base de datos puede distribuirse y replicarse a través de un número de servidores que forman un cluster de servidores, con frecuencia conocido como **arquitectura**

nada compartido, en el cual se enfatiza que ni los discos ni la memoria principal son compartidos por los procesadores. Como alternativa, una base de datos es distribuida, y probablemente replicada, mediante cierto número de sitios dispersos geográficamente. Esta arquitectura se utiliza muy a menudo en bases de datos federadas (Sheth y Larson, 1990).

Réplicas iniciadas por servidores

Por contraste con las réplicas permanentes, las réplicas iniciadas por servidores son copias de un almacén de datos que existe para mejorar el rendimiento, y las cuales son creadas por iniciativa del (propietario del) almacén de datos. Por ejemplo, considere un servidor web ubicado en Nueva York. De manera normal, este servidor puede manejar muy fácilmente peticiones entrantes, pero puede ocurrir que durante un par de días entre una explosión súbita de peticiones desde una ubicación inesperada, alejada del servidor. En ese caso, puede valer la pena instalar cierto número de réplicas temporales en las regiones de donde provienen las peticiones.

El problema de colocar réplicas dinámicamente se está tratando también en los servicios web de hosting. Estos servicios ofrecen una colección (relativamente estática) de servidores dispersos a través de internet que pueden mantener y proporcionar acceso a archivos web pertenecientes a terceras partes. Para proporcionar facilidades óptimas, tales servicios de hosting pueden replicar dinámicamente archivos en los servidores donde dichos archivos son necesarios para mejorar el rendimiento, es decir, en (grupos de) clientes solicitantes cercanos. Sivasubramanian y colaboradores (2004b) proporcionan un tratamiento detallado de la replicación en servicios web de hosting, a lo cual regresaremos en el capítulo 12.

Dado que los servidores de réplicas ya están ubicados, decidir dónde colocar el contenido es más sencillo que en el caso de ubicación de servidores. Un método para implementar la replicación dinámica de archivos en el caso de un servicio web de hosting es descrito por Rabinovich y colaboradores (1999). El algoritmo está diseñado para soportar páginas web, por lo cual supone que las actualizaciones son relativamente raras comparadas con las peticiones de lectura. Al utilizar archivos como la unidad de los datos, el algoritmo funciona de la siguiente manera.

El algoritmo para la replicación dinámica toma en cuenta dos puntos. Primero, la replicación puede ocurrir para reducir la carga en un servidor. Segundo, archivos específicos de un servidor pueden ser migrados o replicados en otros servidores ubicados en la proximidad de los clientes que solicitan mucho esos archivos. En las siguientes páginas, sólo nos concentramos en este segundo punto. También omitimos varios detalles, pero el lector puede encontrarlos en Rabinovich y colaboradores (1999).

Cada servidor da seguimiento a la cantidad de accesos por archivo, y registra de dónde provienen las peticiones de acceso. En particular, se supone que, dado un cliente C, cada servidor puede determinar cuál de los servidores del servicio web de hosting se encuentra más cerca de C. (Tal información puede obtenerse, por ejemplo, a partir de las bases de datos de enrutamiento.) Si el cliente C_1 y el cliente C_2 comparten el mismo servidor "más cercano" P, todas las peticiones de acceso al archivo F en el servidor Q desde C_1 y C_2 se registran conjuntamente en Q como una sola cuenta de acceso $cnt_Q(P,F)$. Esta situación aparece en la figura 7-18.

Cuando en el servidor S la cantidad de peticiones de acceso a un archivo específico F disminuye por debajo de un umbral de eliminación $del(S,F)$, ese archivo puede eliminarse de S. En consecuencia,

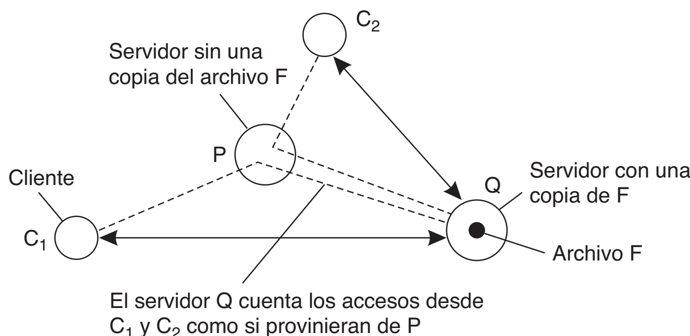


Figura 7-18. Conteo de las peticiones de acceso por parte de diferentes clientes.

el número de réplicas de ese archivo se reduce, lo cual probablemente ocasione mayores cargas de trabajo en otros servidores. Para garantizar que exista al menos una copia de cada archivo, se toman medidas especiales.

Un umbral de replicación *rep*(*S*,*F*), el cual siempre es elegido mayor que el umbral de eliminación, indica si el número de peticiones de acceso a un archivo específico es tan alta que puede valer la pena replicarlo en otro servidor. Si el número de peticiones cae entre los umbrales de eliminación y replicación, sólo se permite que el archivo sea migrado. En otras palabras, en ese caso es importante mantener al menos el mismo número de réplicas de ese archivo.

Cuando un servidor *Q* decide reevaluar la ubicación de los archivos que almacena, verifica el número de accesos por archivo. Si el número total de peticiones de acceso a *F* en *Q* cae por debajo del umbral de eliminación, *del*(*Q*,*F*), el servidor eliminará *F* a menos que ésta sea la última copia. Además, si en algún servidor *P*, $cnt_Q(P,F)$ excede en más de la mitad las peticiones totales de *F* en *Q*, al servidor *P* se le solicita hacerse cargo de *F*. En otras palabras, el servidor *Q* intentará migrar *F* hacia *P*.

La migración del archivo *F* hacia el servidor *P* no siempre es exitosa, por ejemplo, debido a que *P* ya está muy cargado o ya no tiene espacio en disco. En ese caso, *Q* intentará replicar *F* en otros servidores. Por supuesto, la replicación puede ocurrir sólo si el número total de peticiones de acceso a *F* en *Q* excede el umbral de replicación *rep*(*Q*,*F*). El servidor *Q* verifica todos los demás servidores del servicio web de hosting, iniciando con el más alejado. Si, en algún servidor *R*, $cnt_Q(R,F)$ excede cierta fracción de todas las peticiones de acceso a *F* en *Q*, se hace un intento de replicar *F* en *R*.

La replicación iniciada por el servidor continúa para aumentar la popularidad en tiempo, especialmente en el contexto de servicios web de hosting tales como el que acabamos de describir. Observe que mientras se proporcionen garantías de que cada elemento de datos es hospedado por al menos un servidor, esto puede ser suficiente para utilizar sólo la replicación iniciada por el servidor y no tener réplicas permanentes. Sin embargo, las réplicas permanentes aún son útiles como herramientas de respaldo, o para utilizarse como las únicas réplicas que se pueden cambiar para

garantizar la consistencia. Las réplicas iniciadas por el servidor se utilizan para colocar copias de solo lectura cerca de los clientes.

Réplicas iniciadas por el cliente

Una clase importante de réplica es la iniciada por un cliente. Estas réplicas son más comúnmente conocidas como **cachés (cliente)**. En esencia, un caché es una herramienta de almacenaje local utilizada por un cliente para almacenar temporalmente una copia de los datos que ha solicitado. En principio, la administración del caché se le deja completamente al cliente. El almacén de datos desde donde éstos son traídos no tiene nada que ver con mantener consistentes los datos cacheados. Sin embargo, como veremos, hay muchas ocasiones en que el cliente puede confiar en la intervención del almacén de datos para que le informe cuándo se han devuelto los datos cacheados obsoletos.

Los cachés cliente sólo se utilizan para mejorar el tiempo de acceso a los datos. Normalmente, cuando un cliente quiere acceder a ciertos datos se conecta con la copia más cercana del almacén de datos desde donde trae los datos que quiere leer, o a donde almacena los datos que acaba de modificar. Cuando la mayoría de las operaciones sólo involucra datos de lectura, es posible mejorar el rendimiento si se permite que el cliente almacene los datos solicitados en un caché cercano. Dicho caché podría ubicarse en la máquina del cliente, o en otra máquina de la misma red de área local del cliente. La próxima vez que necesite leer los datos, el cliente puede simplemente traerlos desde su caché local. Este esquema funciona bien siempre y cuando los datos traídos no se hayan modificado.

Por lo general, los datos se mantienen en caché un tiempo limitado, por ejemplo, para prevenir el uso de datos extremadamente viejos, o simplemente para dar espacio a otros datos. Siempre que los datos pueden traerse desde el caché local, se dice que ocurrió un **acierto del caché**. Para mejorar el número de aciertos del caché, los cachés pueden compartirse entre clientes. La suposición subyacente es que una petición de datos del cliente C_1 también puede ser útil para una petición de otro cliente cercano C_2.

El que esta suposición sea correcta depende en gran medida del tipo de almacén de los datos. Por ejemplo, en sistemas de archivo tradicionales, los archivos de datos rara vez se comparten completamente (vea Muntz y Honeyman, 1992; y Blaze, 1993), lo cual se traduce en un caché compartido inútil. Asimismo, resulta que el uso de cachés para compartir datos está perdiendo terreno, en parte debido a las mejoras en la red y al desempeño del servidor. En cambio, los esquemas de replicación iniciada por el servidor se están volviendo más efectivos.

Ubicar cachés cliente es relativamente sencillo: por lo general, un caché se coloca en la misma máquina del cliente, o en otra máquina compartida por clientes de la misma red de área local. Sin embargo, en algunos casos, administradores de sistema introducen niveles adicionales de cacheo colocando un caché compartido entre varios departamentos u organizaciones, o incluso colocando un caché compartido para toda una región, como una provincia o un país.

Otro método es colocar servidores (caché) en puntos específicos de una red de área amplia, y dejar que el cliente localice al servidor más cercano. Cuando el servidor es localizado, se le puede solicitar mantener copias de los datos que el cliente pidió con anterioridad desde algún otro lado,

según describen Noble y colaboradores (1999). Retomaremos el tema del cacheo en este capítulo cuando expliquemos los protocolos de consistencia.

7.4.3 Distribución de contenido

La administración de réplicas también trata con la propagación (actualización) de contenido hacia los servidores réplica significativos.

Estado *versus* operaciones

Un punto importante de diseño se refiere a lo que en realidad va a propagarse. Básicamente existen tres posibilidades:

1. Propagar solamente la notificación de una actualización.
2. Transferir datos de una copia a otra.
3. Propagar la operación de actualización hacia otras copias.

Propagar una notificación es lo que hacen los **protocolos de invalidación**. En un protocolo de invalidación, se les informa a otras copias que ha ocurrido una actualización, y que los datos que ellas contienen ya no son válidos. La invalidación puede especificar qué parte del almacén de datos se ha actualizado, de modo tal que, en realidad, sólo parte de una copia no es válida. El punto importante es que no se propaga más de una notificación. Siempre que se solicita una operación o una copia invalidada, dicha copia generalmente necesita actualizarse primero, de acuerdo con el modelo específico de consistencia que va a soportarse.

La principal ventaja de los protocolos de invalidación es que utilizan muy poco ancho de banda de la red. La única información que necesita transferirse es una especificación de cuáles datos ya no son válidos. Dichos protocolos generalmente funcionan mejor cuando hay muchas operaciones de actualización, comparado con las operaciones de lectura; esto es, la relación lectura-escritura es relativamente pequeña.

Por ejemplo, consideremos un almacén de datos en el que las actualizaciones se propagan enviando los datos modificados hacia todas las réplicas. Si el tamaño de los datos modificados es grande, y las actualizaciones ocurren frecuentemente, comparado con las operaciones de lectura, podríamos enfrentar la situación en que dos actualizaciones ocurren una después de otra, sin que alguna operación de lectura se realice entre ellas. En consecuencia, la propagación de la primera actualización hacia todas las réplicas es efectivamente inútil, ya que será sobreescrita por la segunda actualización. En cambio, enviar una notificación de que los datos se modificaron sería más eficiente.

La segunda alternativa es transferir los datos modificados entre las réplicas, y es útil cuando la relación lectura-escritura es relativamente grande. En ese caso, la probabilidad de que una actualización resulte efectiva, en el sentido de que los datos modificados serán leídos antes de que ocurra la siguiente actualización, es grande. En lugar de propagar los datos modificados, también es posible registrar los cambios y transferir sólo esos registros para ahorrar ancho de banda. Además, a menudo

las transferencias se juntan en el sentido de que se empacan varias modificaciones en un solo mensaje, evitando así la sobrecarga de comunicación.

El tercer método es no transferir ninguna modificación de datos, sino indicarle a cada réplica qué operación de actualización debe realizar (y enviar solamente los valores de los parámetros que esas operaciones necesitan). Este método, también conocido como **replicación activa**, supone que cada réplica es representada por un proceso capaz de mantener "activamente" actualizados sus datos asociados mediante la realización de operaciones (Schneider, 1990). El beneficio principal de la replicación activa es que las actualizaciones pueden propagarse con costos mínimos de ancho de banda debido a que el tamaño de los parámetros asociados con una operación es relativamente pequeño. Además, las operaciones pueden ser arbitrariamente complejas, lo cual permite mejoras posteriores para mantener las réplicas consistentes. Por otra parte, es posible que cada réplica necesite un mayor poder de procesamiento, en especial cuando las operaciones son relativamente complejas.

Protocolos pull *versus* protocolos push

Otro tema de diseño es si las actualizaciones se insertan o se extraen. En un **método basado en push**, también conocido como **protocolos basados en servidor**, las actualizaciones se propagan hacia otras réplicas sin que éstas lo soliciten. Los métodos basados en push con frecuencia se utilizan entre réplicas permanentes y las iniciadas por servidor, pero también pueden utilizarse para insertar actualizaciones en cachés cliente. Los protocolos basados en servidor se aplican cuando las réplicas necesitan mantener un grado de consistencia relativamente alto. En otras palabras, es necesario que las réplicas se mantengan idénticas.

Esta necesidad de un alto grado de consistencia se relaciona con el hecho de que las réplicas permanentes y las iniciadas por servidor, así como los grandes cachés, con frecuencia son compartidos por muchos clientes quienes, a su vez, básicamente realizan operaciones de lectura. Por tanto, en cada réplica, la relación lectura-actualización es relativamente alta. En estos casos, los protocolos basados en push son eficientes en el sentido de que cada actualización insertada puede utilizarse por uno o más lectores. Además, los protocolos basados en push propician que los datos consistentes estén disponibles de inmediato conforme son solicitados.

Por contraste, en un **método basado en pull**, un servidor o un cliente solicita a otro servidor que le envíe cualquier actualización que tenga hasta el momento. Los protocolos basados en pull, también conocidos como **protocolos basados en el cliente**, con frecuencia son utilizados por cachés cliente. Por ejemplo, una estrategia común, aplicada a los cachés web, es verificar primero si los elementos de datos cacheados aún están actualizados. Cuando un caché recibe una petición de elementos que aún están localmente disponibles, el caché verifica con el servidor web original si esos elementos de datos han sido modificados a partir de que fueron cacheados. En caso de una modificación, los datos modificados se transfieren primero al caché, y después se devuelven al cliente solicitante. Si no ocurrió modificación alguna, los datos cacheados son devueltos. En otras palabras, el cliente sondea al servidor para ver si se necesita una actualización.

Un método basado en pull es eficiente cuando la relación lectura-actualización es relativamente baja. En general, éste es el caso con cachés cliente (no compartidos), los cuales sólo tienen un cliente. Sin embargo, incluso cuando un caché es compartido por muchos clientes, un método basado en pull también puede demostrar su eficiencia cuando los elementos de datos cacheados son

compartidos. La desventaja principal de una estrategia basada en pull, comparada con el método push, es que el tiempo de respuesta se incrementa en el caso de una pérdida de caché.

Cuando se comparan soluciones basadas en push con las basadas en pull, hay muchos sacrificios por hacer, según muestra la figura 7-19. Por simplicidad, considere un sistema cliente-servidor que consta de un solo servidor no distribuido, y varios procesos cliente, cada uno con su propio caché.

Cuestión	Basado en push	Basado en pull
Estado en el servidor	Lista de réplicas cliente y cachés	Ninguno
Mensajes enviados	Actualiza (y posiblemente trae la actualización después)	Sondea y actualiza
Tiempo de respuesta en el cliente	Inmediato (o busca el tiempo de actualización)	Busca el tiempo de actualización

Figura 7-19. Comparación entre los protocolos basados en push y los basados en pull para el caso de sistemas de un solo servidor y varios clientes.

Un punto importante es que en los protocolos basados en push, el servidor necesita dar seguimiento a todos los cachés cliente. Además del hecho de que los servidores de estado son con frecuencia menos tolerantes a las fallas, como explicamos en el capítulo 3, dar seguimiento a todos los cachés cliente puede producir una sobrecarga considerable en el servidor. Por ejemplo, en un método basado en push, un servidor web puede necesitar dar seguimiento a decenas de miles de cachés cliente. Cada vez que se actualiza una página web, el servidor necesitará revisar su lista de cachés cliente, mantener una copia de esa página, y posteriormente propagar la actualización. Peor aún, si un cliente purga una página debido a la falta de espacio, tiene que informarlo al servidor, ello da pie a más comunicación.

Los mensajes que es necesario enviar entre un cliente y el servidor también difieren. En un método basado en push, la única comunicación es que el servidor envía actualizaciones a cada cliente. Cuando las actualizaciones en realidad sólo son invalidaciones, se necesita comunicación adicional por parte del cliente para traer los datos modificados. En un método basado en pull, el cliente tendrá que sondear al servidor y, si es necesario, traer los datos modificados.

Por último, el tiempo de respuesta en el cliente también es diferente. Cuando un servidor introduce datos modificados en el caché cliente, queda claro que el tiempo de respuesta del lado del cliente es cero. Cuando se introducen las invalidaciones, el tiempo de respuesta es el mismo que en el método basado en pull, y se determina por el tiempo que toma traer los datos modificados desde el servidor.

Estas desventajas han dado lugar a una forma híbrida de propagación de actualizaciones basada en contratos. Un **contrato** es una promesa del servidor de que introducirá actualizaciones al cliente por el tiempo especificado. Cuando un contrato expira, el cliente es forzado a sondear al servidor en cuanto a actualizaciones y a extraer los datos modificados si es necesario. Una alternativa es que el cliente solicite un nuevo contrato para introducción de actualizaciones cuando el anterior expire.

Los contratos fueron presentados originalmente por Gray y Cheriton (1989). Ellos proporcionaron un mecanismo conveniente para el cambio dinámico entre una estrategia basada en push y

otra basada en pull. Duvvuri y colaboradores (2003) describen un sistema flexible de contratos que permite que el tiempo de vencimiento sea adaptado dinámicamente, de acuerdo con los diferentes criterios de contratación. Ellos distinguen los tres siguientes tipos de contrato. (Observe que en todos los casos, las actualizaciones son introducidas por los servidores siempre y cuando el contrato no haya vencido.)

Primero, los contratos basados en el tiempo se desprenden de los elementos de datos de acuerdo con la última vez que se modificó el elemento. La suposición subyacente es que es de esperarse que los datos que no han sido modificados durante un largo tiempo permanezcan así durante un tiempo más. Esta suposición ha demostrado ser razonable en el caso de datos basados en la web. Al garantizar contratos de larga duración a los elementos de datos que esperan permanecer sin modificaciones, el número de mensajes de actualización puede reducirse de manera importante en comparación con el caso en que todos los contratos tienen el mismo tiempo de vencimiento.

Otro criterio de contratación es la frecuencia con que un cliente específico solicita que su copia cacheada se actualice. Con contratos basados en la frecuencia de renovación, un servidor manejará un contrato de larga duración con un cliente cuyo caché necesita ser refrescado con frecuencia. Por otra parte, un cliente que sólo ocasionalmente solicita un elemento específico de datos será manejado con un contrato de corto plazo para ese elemento. El efecto de esta estrategia es que el servidor da seguimiento básicamente sólo a aquellos clientes entre los que sus datos son muy populares; además, a esos clientes se les ofrece un alto grado de consistencia.

El último criterio es el de la sobrecarga de espacio-estado en el servidor. Cuando el servidor advierte que gradualmente se está sobrecargando, disminuye el tiempo de vencimiento de los nuevos contratos y los distribuye a los clientes. El efecto de esta estrategia es que el servidor debe dar seguimiento a unos cuantos clientes, ya que los contratos expiran más rápidamente. En otras palabras, el servidor cambia dinámicamente a un modo más desahogado de operación, con lo cual se descarga a sí mismo para poder manejar las peticiones de manera más eficiente.

Difusión simple *versus* multidifusión

Relacionado con introducir o extraer actualizaciones, se encuentra el decidir si se utiliza la difusión simple o la multidifusión. En la comunicación de difusión simple, cuando un servidor que es parte del almacén de datos envía su actualización a otros N servidores, lo hace enviando N mensajes separados, uno a cada servidor. Con la multidifusión, la red subyacente se encarga de enviar eficientemente un mensaje a varios destinatarios.

En muchos casos, resulta más barato utilizar las herramientas de multidifusión disponibles. Una situación extrema es cuando todas las réplicas se encuentran en la misma red de área local y el hardware de transmisión está disponible. En ese caso, transmitir o multidifundir un mensaje no resulta más caro que un solo mensaje punto a punto. Entonces la difusión simple de actualizaciones resultaría menos eficiente.

La multidifusión puede ser eficientemente combinada con un método basado en push para propagar actualizaciones. Cuando los dos tipos de difusión se integran cuidadosamente, el servidor que decide introducir sus actualizaciones en otros servidores simplemente utiliza un solo grupo de multidifusión para enviar sus actualizaciones. Por contraste, con un método basado en pull, generalmente sólo hay un cliente o un servidor que solicita actualizar su copia. En ese caso, la difusión simple puede ser la solución más eficiente.

7.5 PROTOCOLOS DE CONSISTENCIA

Hasta el momento, nos hemos concentrado principalmente en varios modelos de consistencia, y en cuestiones generales de diseño para protocolos de consistencia. En esta sección nos concentramos en la implementación real de los modelos de consistencia, analizando diversos protocolos de consistencia. Un **protocolo de consistencia** describe la implementación de un modelo específico de consistencia. Seguiremos la organización de nuestra explicación sobre modelos de consistencia analizando primero los modelos centrados en los datos, y después los protocolos para modelos centrados en el cliente.

7.5.1 Consistencia continua

Como parte de su trabajo sobre consistencia continua, Yu y Vahdat desarrollaron varios protocolos para abordar las tres formas de consistencia. A continuación, consideraremos brevemente varias soluciones y, por claridad, omitiremos los detalles.

Límites para la desviación numérica

Primero nos concentramos en una solución para mantener limitada la desviación numérica. De nuevo, nuestro objetivo no es entrar en detalles para cada protocolo, sino dar una idea general. Los detalles para limitar la desviación numérica pueden encontrarse en Yu y Vahdat (2000b).

Nos enfocaremos en escrituras sobre un solo elemento de datos x. Cada escritura $W(x)$ tiene una ponderación asociada que representa el valor numérico mediante el cual x es actualizada, y se denota como *ponderación* ($W(x)$), o simplemente *ponderación* (W). Por simplicidad, suponemos que *ponderación* (W) > 0. Cada escritura W es inicialmente enviada a uno de los N servidores de réplicas disponibles, y en ese caso, dicho servidor se vuelve el origen de la escritura y se denota como *origen*(w). Si consideramos el sistema en un punto específico de tiempo, veremos varias escrituras enviadas que aún necesitan propagarse a todos los servidores. Con este fin, cada servidor S_i dará seguimiento a un registro L_i de escrituras que ha realizado en su propia copia local de x.

Sean $TW[i,j]$ las escrituras ejecutadas por el servidor S_i que se originaron desde el servidor S_j:

$$TW[i,j] = \sum \{ponderación(W) \mid origen(W) = S_j \ \& \ W \in L_i\}$$

Observe que $TW[i,j]$ representa el grupo de escrituras enviadas a S_i. Nuestro objetivo es que para cualquier tiempo t, el valor actual v_i en el servidor S_i se desvíe dentro de los límites desde el valor real $v(t)$ de x. Este valor real está determinado completamente por todas las escrituras enviadas. Es decir, si $v(0)$ es el valor inicial de x, entonces

$$v(t) = v(0) + \sum_{k=1}^{N} TW[k,k]$$

y

$$v_i = v(0) + \sum_{k=1}^{N} TW[i,k]$$

Observe que $v_i \leq v(t)$. Concentrémonos sólo en desviaciones absolutas. En particular, para cada servidor S_i, asociamos un límite superior δ_i tal que necesitamos hacer cumplir que:

$$v(t) - v_i \leq \delta_i$$

Las escrituras enviadas a un servidor S_i necesitarán propagarse a todos los demás servidores. Hay diferentes formas para hacer esto, pero en general un protocolo epidémico permitirá una rápida diseminación de actualizaciones. En cualquier caso, cuando un servidor S_i propaga una escritura que se origina en S_j hacia S_k, este último será capaz de aprender el valor $TW[i,j]$ en el momento en que se envió la escritura. En otras palabras, S_k puede mantener una **vista** $TW_k[i,j]$ de lo que cree que S_i tendrá como valor para $TW[i,j]$. Resulta evidente que,

$$0 \leq TW_k[i,j] \leq TW[i,j] \leq TW[j,j]$$

La idea completa es que cuando el servidor S_k advierte que S_i no ha mantenido el ritmo de las actualizaciones enviadas a S_k, reenvía las escrituras desde su registro hacia S_i. Este reenvío efectivamente *avanza* la vista $TW_k[i,k]$ que S_k tiene de $TW[i,k]$, haciendo pequeña la desviación $TW[i,k] - TW_k[i,k]$. En particular, S_k avanza su vista sobre $TW[i,k]$ cuando una aplicación envía una nueva escritura que incrementaría $TW[k,k] - TW_k[i,k]$ más allá de S_i / $(N - 1)$. Dejamos esto como ejercicio para mostrar que el avance siempre garantiza que $v(t) - v_i \leq \delta_i$.

Límites para desviaciones descontinuadas

Hay muchas formas de mantener réplicas descontinuadas dentro de límites específicos. Un método sencillo es dejar que el servidor S_k mantenga un reloj vectorial de tiempo real RVC_k, donde $RVC_k[i] = T(i)$ significa que S_k ha visto todas las escrituras enviadas a S_i al tiempo $T(i)$. En este caso, suponemos que cada escritura enviada es registrada por su servidor origen, y que $T(i)$ denota el tiempo *local* en S_i.

Si los relojes entre los servidores de réplicas están aproximadamente sincronizados, entonces un protocolo aceptable para limitar la discontinuidad sería el siguiente. Siempre que el servidor S_k advierte que $T(k) - RVC_k[i]$ está por exceder un límite especificado, simplemente comienza a introducir escrituras que se originaron desde S_i con un registro de tiempo posterior a $RVC_k[i]$.

Observe que en este caso, un servidor de réplicas es responsable de mantener actualizada su copia de x con respecto a las escrituras que se han distribuido en alguna otra parte. Por contraste, cuando se mantienen límites numéricos, seguimos un método push dejando que un servidor origen mantenga actualizadas las réplicas reenviando las escrituras. El problema con la inserción de escrituras, en el caso de la discontinuidad, es que no hay garantías de consistencia cuando no se sabe con anterioridad cuál será la máxima propagación. Esta situación mejora, de alguna manera, con la extracción de actualizaciones, ya que varios servidores pueden ayudar a mantener fresca (actualizada) la copia de x de un servidor.

Límites para desviaciones de ordenamiento

Recuerde que en la consistencia continua, las desviaciones de ordenamiento son causadas por el hecho de que un servidor de réplicas aplica tentativamente actualizaciones que le han sido enviadas. Como resultado, cada servidor tendrá una cola local de escrituras tentativas para las cuales el orden en que van a aplicarse a la copia local de *x* aún debe ser determinado. La desviación de ordenamiento se limita especificando la longitud máxima de la cola de escrituras tentativas.

Por tanto, detectar cuándo es necesario reforzar la consistencia de ordenamiento es simple: cuando la longitud de esta cola local excede una longitud máxima especificada. En ese punto, un servidor ya no aceptará ninguna escritura enviada recientemente, pero sí aceptará confirmar escrituras tentativas negociando con otros servidores el orden en que deben ejecutarse sus escrituras. En otras palabras, necesitamos reforzar un ordenamiento global consistente de escrituras tentativas. Hay muchas maneras de hacer esto, pero resulta que, en la práctica, se utilizan los protocolos también conocidos como basados en primarias o en quórum. A continuación explicaremos estos protocolos.

7.5.2 Protocolos basados en primarias

En la práctica, vemos que las aplicaciones distribuidas siguen modelos de consistencia que son relativamente fáciles de entender. Estos modelos incluyen a los que limitan la desviación de discontinuidad, y en menor medida a los que limitan las desviaciones numéricas. Cuando se trata de modelos que manejan el ordenamiento consistente de operaciones, o consistencia secuencial, son más populares aquellos en los que las operaciones pueden agruparse mediante candados o transacciones.

Tan pronto como los modelos de consistencia se vuelven ligeramente difíciles de entender para los desarrolladores de aplicaciones, vemos que son ignorados aun cuando el rendimiento pudiera mejorarse. El resultado final es que si la semántica de un modelo de consistencia no resulta intuitivamente clara, los desarrolladores la pasarán mal para construir aplicaciones correctas. La sencillez es muy apreciada (y tal vez justificada).

En el caso de la consistencia secuencial, resulta que prevalecen los protocolos basados en primarias. En estos protocolos, cada elemento de datos *x* del almacén de datos tiene una primaria asociada, la cual es responsable de coordinar las operaciones de escritura sobre *x*. Es posible diferenciar si la primaria está fija en un servidor remoto o si las operaciones de escritura pueden realizarse localmente después de trasladar la primaria al proceso en donde se inició la operación de escritura. Demos un vistazo a esta clase de protocolos.

Protocolos de escritura remota

El protocolo más sencillo basado en primarias que soporta la replicación es aquél en el que todas las operaciones de escritura necesitan remitirse a un solo servidor fijo. Tales esquemas también se conocen como **protocolos primarios de respaldo** (Budhiraja y cols., 1993). Un protocolo primario de respaldo funciona como nos muestra la figura 7-20. Un proceso que espera para realizar una operación de escritura sobre un elemento de datos *x*, remite esa operación al servidor primario de *x*.

La primaria realiza la actualización en su copia local de *x*, y posteriormente remite la actualización a los servidores de respaldo. Cada servidor de respaldo también realiza la actualización, y envía un acuse de recibo de vuelta a la primaria. Cuando todos los respaldos han actualizado su copia local, la primaria envía un acuse de vuelta al proceso inicial.

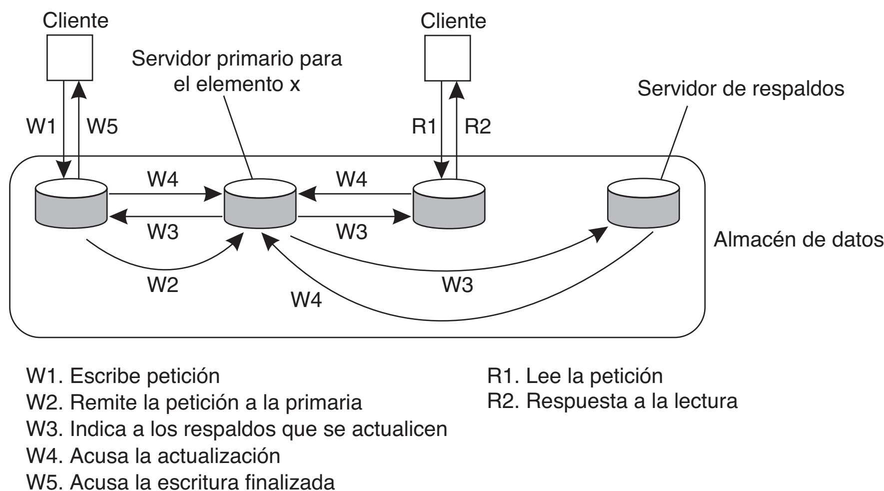


Figura 7-20. Principio de un protocolo primario de respaldo.

Un problema potencial de rendimiento con este esquema es que puede pasar mucho tiempo antes de que al proceso que inició la actualización se le permita continuar. En efecto, una actualización es implementada como una operación de bloqueo. Una alternativa es utilizar un método de no bloqueo. Tan pronto como la primaria ha actualizado su copia local de *x*, ésta devuelve un acuse; después de lo cual indica a los servidores de respaldos que también realicen la actualización. Budhiraja y Marzullo (1992) explican los protocolos primarios de respaldo de no bloqueo.

El problema principal con los protocolos primarios de respaldo de no bloqueo tiene que ver con la tolerancia a las fallas. En un esquema de bloqueo, el proceso cliente sabe con certeza que la operación de actualización es respaldada por varios servidores. Éste no es el caso con una solución de no bloqueo. La ventaja, por supuesto, es que las operaciones de escritura pueden ser considerablemente rápidas. En el siguiente capítulo retomaremos ampliamente las cuestiones de tolerancia a las fallas.

Los protocolos primarios de respaldo proporcionan una implementación directa de la consistencia secuencial, ya que la primaria puede ordenar todas las escrituras entrantes en un orden de tiempo globalmente único. Es evidente que todos los procesos ven a todas las operaciones de escritura en el mismo orden, independientemente del servidor de respaldos que utilicen para realizar las operaciones de lectura. Además, con los protocolos de bloqueo, los procesos siempre verán los efectos de su operación de escritura más reciente (observe que esto no puede garantizarse con un protocolo de no bloqueo sin tomar medidas especiales).

Protocolos de escritura local

Una variante de los protocolos primarios de respaldo es un protocolo en el que la copia primaria migra entre procesos que desean realizar una operación de escritura. Como antes, siempre que un proceso quiere actualizar el elemento *x*, éste localiza la copia primaria de *x* y posteriormente la lleva a su propia ubicación, como lo muestra la figura 7-21. La principal ventaja de este método es que varias operaciones sucesivas de escritura pueden realizarse localmente, mientras que los procesos de lectura aún pueden acceder a su copia local. Sin embargo, tal mejora sólo puede lograrse si un protocolo de no bloqueo es seguido por las actualizaciones que se propagan a las réplicas después de que la primaria ha terminado de realizar localmente las actualizaciones.

W1. Escribe la petición
W2. Lleva el elemento x a la nueva primaria
W3. Acusa la escritura finalizada
W4. Indica a los respaldos que se actualicen
W5. Acusa la actualización

R1. Lee la petición
R2. Respuesta a la lectura

Figura 7-21. Protocolo primario de respaldo en el que la primaria migra hacia el proceso que quiere realizar una actualización.

Este protocolo primario de respaldo de escritura local también puede aplicarse a computadoras móviles que son capaces de operar desconectadas. Antes de desconectarla, la computadora móvil se vuelve el servidor primario para cada elemento de datos que espera actualizar. Mientras es desconectada, todas las operaciones de actualización se llevan a cabo localmente en lo que otros procesos aún pueden realizar operaciones de lectura (pero no actualizaciones). Después, cuando se conecta de nuevo, las actualizaciones se propagan desde la primaria hacia los respaldos, llevando al almacén de datos de nuevo a un estado consistente. En el capítulo 11 retomaremos la operación en modo de desconexión, cuando expliquemos los sistemas de archivo distribuidos.

Como una última variante de este esquema, los protocolos basados en primarias de escritura local y de no bloqueo también son utilizados por sistemas de archivo distribuidos en general. En este caso, puede haber un servidor central fijo a través del cual, normalmente, ocurren todas las operaciones de escritura, tal como en el caso de la primaria de respaldo de escritura remota. Sin

embargo, de modo temporal el servidor permite que una de las réplicas realice una serie de actualizaciones locales, ya que esto puede incrementar considerablemente el rendimiento. Cuando el servidor de réplicas termina, las actualizaciones se propagan hacia el servidor central, desde donde se distribuyen hacia los otros servidores de réplicas.

7.5.3 Protocolos de escritura replicados

En los protocolos de escritura replicados, las operaciones de escritura pueden realizarse en varias réplicas en lugar de sólo en una, como en el caso de réplicas basadas en primarias. Es posible diferenciar la replicación activa, en la que una operación se remite a todas las réplicas, y los protocolos de consistencia basados en la mayoría de votos.

Replicación activa

En la replicación activa, cada réplica tiene un proceso asociado que realiza operaciones de actualización. Por contraste con otros protocolos, generalmente las actualizaciones se propagan mediante la operación de escritura que causa la actualización. En otras palabras, la operación se envía a cada réplica. Sin embargo, también es posible enviar la actualización, como explicamos antes.

Un problema con la replicación activa es que las operaciones deben realizarse en el mismo orden en cualquier parte. En consecuencia, lo que se necesita es un mecanismo de multidifusión totalmente ordenado. Tal multidifusión puede implementarse mediante los relojes lógicos de Lamport, como explicamos en el capítulo anterior. Por desgracia, esta implementación de la multidifusión no se escala bien en sistemas distribuidos. Como alternativa, el ordenamiento total puede lograrse utilizando un coordinador central, también llamado **secuenciador**. Un método es remitir primero cada operación al secuenciador, el cual le asigna un número de secuencia único y posteriormente remite la operación a todas las réplicas. Las operaciones se realizan según el orden de sus números de secuencia. Resulta claro que esta implementación de multidifusión totalmente ordenada se parece mucho a los protocolos de consistencia basados en primarias.

Observe que utilizar un secuenciador no resuelve el problema de escalabilidad. De hecho, si se necesita la multidifusión totalmente ordenada, puede requerirse también una combinación de multidifusión simétrica utilizando registros de tiempo Lamport y secuenciadores. Dicha solución se describe en Rodrigues y colaboradores (1996).

Protocolos basados en quórum

Un método diferente para soportar escrituras replicadas es utilizar la **votación** como la propuso originalmente Thomas (1979) y fue generalizada por Gifford (1979). La idea básica es requerir a los clientes que soliciten y adquieran el permiso de varios servidores antes de leer o escribir un elemento de datos replicado.

Como un ejemplo sencillo del funcionamiento del algoritmo, considere un sistema distribuido de archivos y suponga que un archivo se replica en N servidores. Podríamos implementar una regla

que establezca que para actualizar un archivo, un cliente primero debe contactar al menos a la mitad de los servidores más uno (la mayoría) y hacerlos aceptar que se realice la actualización. Una vez que lo acuerden, el archivo es cambiado y se asocia un nuevo número de versión al archivo resultante. El número de versión se utiliza para identificar la versión del archivo, y es el mismo para todos los archivos actualizados recientemente.

Para leer un archivo replicado, un cliente debe contactar también al menos a la mitad de los servidores más uno y solicitarles le envíen los números de versión asociados con el archivo. Si todos los números de versión son los mismos, ésta debe ser la versión más reciente porque intentar actualizar sólo a los servidores restantes fallaría ya que no hay suficientes de ellos.

Por ejemplo, si hay cinco servidores y un cliente determina que tres de ellos tienen la versión 8, es imposible que los otros dos tengan la versión 9. Después de todo, cualquier actualización exitosa de la versión 8 a la 9 requiere que los tres servidores lo acuerden, y no sólo dos.

En realidad, el esquema de Gifford es algo más general que esto. En él, para leer un archivo del que existen N réplicas, un cliente necesita reunir un **quórum de lectura**, una colección cualquiera de N_R servidores o más. De manera similar, para modificar un archivo, se requiere un **quórum de escritura** de al menos N_W servidores. Los valores de N_R y N_W están sujetos a las siguientes dos restricciones:

1. $N_R + N_W > N$
2. $N_W > N/2$

La primera restricción se utiliza para evitar conflictos de lectura-escritura, mientras que la segunda evita conflictos de escritura-escritura. Sólo después de que el número adecuado de servidores ha acordado participar, un archivo puede leerse o escribirse.

Para ver cómo funciona este algoritmo, consideremos la figura 7-22(a), en la que $N_R = 3$ y $N_W = 10$. Imagine que el quórum de escritura más reciente consiste en los 10 servidores, C a L. Todos obtienen la nueva versión y el nuevo número de versión. Cualquier quórum de lectura subsiguiente de tres servidores tendrá que contener al menos un miembro de este conjunto. Cuando el cliente vea los números de versión, sabrá cuál es la más reciente y la tomará.

En las figuras 7-22(b) y (c), vemos dos ejemplos más. En la figura 7-22(b) puede ocurrir un conflicto de escritura-escritura ya que $N_W \leq N/2$. En particular, si un cliente elige $\{A,B,C,E,F,G\}$ como su conjunto de escritura y otro cliente elige $\{D,H,I,J,K,L\}$ como su conjunto de escritura, entonces claramente caeremos en el problema de que serán aceptadas dos actualizaciones sin detectar que en realidad están en conflicto.

La situación que muestra la figura 7-22(c) es especialmente interesante, ya que establece N_R en uno, lo cual hace posible leer un archivo replicado al encontrar cualquier copia y usarla. Sin embargo, el precio pagado por este buen rendimiento de lectura es que las actualizaciones de lectura necesitan adquirir todas las copias. Este esquema, en general, se conoce como **lee uno, escribe todo** (**ROWA**, por sus siglas en inglés). Existen diversas variantes de los protocolos de replicación basados en quórum. Un buen panorama se encuentra en Jalote (1994).

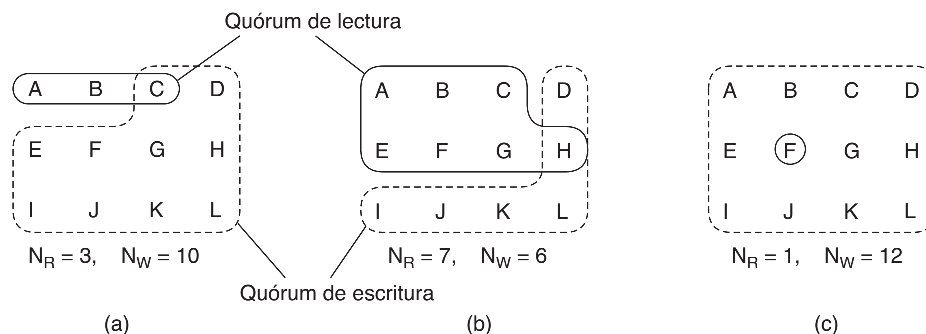


Figura 7-22. Tres ejemplos del algoritmo de votación. (a) Elección correcta del conjunto de lectura y escritura. (b) Elección que puede derivar en conflictos escritura-escritura. (c) Elección correcta, conocida como ROWA (lee uno, escribe todo).

7.5.4 Protocolos de coherencia de caché

Los cachés forman un caso especial de replicación en el sentido de que, generalmente, son controlados por los clientes en lugar de por los servidores. Sin embargo, los protocolos de coherencia de caché, los cuales garantizan la consistencia del caché con las réplicas iniciadas por el servidor son, en principio, no muy diferentes de los protocolos de consistencia que hemos explicado hasta aquí.

Hoy en día se lleva a cabo mucha investigación sobre el diseño y la implementación de cachés, especialmente en el contexto de sistemas multiproceso de memoria compartida. Muchas soluciones están basadas en el soporte del hardware subyacente, por ejemplo, al asumir que se puede llevar a cabo el “snooping” o multitransmisión. En el contexto de los sistemas distribuidos basados en middleware que se construyen en la cima de sistemas operativos de propósito general, las soluciones basadas en software son más interesantes. En este caso, con frecuencia se mantienen dos criterios separados para clasificar los protocolos de cacheo (Min y Baer, 1992; Lilja, 1993; y Tartalja y Milutinovic, 1997).

Primero, las soluciones de cacheo pueden diferir en su **estrategia de detección de coherencia**, esto es, *cuándo* se detectan en realidad las inconsistencias. En las soluciones estáticas, asumimos que el compilador realiza el análisis necesario previo a la ejecución, y para determinar efectivamente cuál dato pudiera provocar inconsistencias debido a que pueda ser cacheado. El compilador simplemente inserta instrucciones que evitan inconsistencias. Por lo general, las soluciones dinámicas se aplican en los sistemas distribuidos que explicamos aquí. En estas soluciones, las inconsistencias se detectan en tiempo de ejecución. Por ejemplo, se realiza una verificación con el servidor para ver si los datos cacheados fueron modificados desde que fueron cacheados.

En el caso de bases de datos distribuidas, los protocolos dinámicos basados en detección pueden clasificarse todavía más, considerando cuándo exactamente se realiza la detección durante una transacción. Franklin y colaboradores (1997) diferencian los siguientes tres casos. Primero, cuando durante una transacción se accede a un elemento de datos cacheado, el cliente requiere verificar si

dicho elemento de datos es aún consistente con la versión almacenada en un servidor (posiblemente replicado). La transacción no puede proceder con el uso de la versión cacheada sino hasta que su consistencia se ha validado de manera definitiva.

Una segundo método, optimista, es permitir que la transacción proceda mientras se lleva a cabo la verificación. En este caso, asumimos que los datos cacheados se actualizaron al comenzar la transacción. Si posteriormente dicha suposición resulta ser falsa, se tendrá que abortar la transacción.

El tercer método es verificar si los datos cacheados están actualizados sólo cuando se confirme la transacción. Este método es comparable con el esquema optimista de control de monedas explicado en el capítulo anterior. En efecto, la transacción comienza a operar justo en los datos cacheados y simplemente espera que suceda lo mejor. Después de todo el trabajo que ha realizado, verifica la consistencia de los datos a los que accede. Cuando se utilizan datos antiguos (descontinuados), se aborta la transacción.

Otro problema de diseño con los protocolos de coherencia de caché es la **estrategia de reforzamiento de coherencia**, la cual determina la *forma* en que se mantienen consistentes los cachés con el número de copias almacenadas en los servidores. La solución más simple es deshabilitar todos los datos almacenados que se van a cachear. En vez de eso, los datos compartidos se mantienen sólo en los servidores, los cuales mantienen la consistencia mediante uno de los protocolos que explicamos antes. A los clientes se les permite cachear sólo datos privados. Desde luego, esta solución solamente puede ofrecer mejoras limitadas al rendimiento.

Para cachear datos compartidos existen dos métodos que refuerzan la coherencia del caché. El primer método es permitir que el servidor envíe una invalidación a todos los cachés cada vez que se modifica un elemento de datos. El segundo es simplemente propagar la actualización. Muchos de los sistemas de cacheo utilizan uno de estos dos esquemas. Elegir dinámicamente entre enviar invalidaciones o actualizaciones a veces es soportado por bases de datos cliente-servidor (Franklin y cols., 1997).

Por último, también necesitamos considerar lo que sucede cuando un proceso modifica datos en el caché. Cuando se utilizan cachés de sólo lectura, solamente los operadores pueden realizar las operaciones de actualización, las cuales subsecuentemente siguen algún protocolo de distribución para asegurar que las actualizaciones se propaguen hacia los cachés. En muchos casos, se sigue un método basado en pull. En este caso, un cliente detecta que su caché está descontinuado, y solicita al servidor una actualización.

Un método alternativo es permitir a los clientes modificar directamente los datos cacheados, y reenviar la actualización hacia los servidores. Este método de escritura se aplica en **cachés de escritura**, los cuales con frecuencia se utilizan en sistemas de archivo distribuidos. En efecto, el cacheo de escritura se parece al protocolo de escritura local basado en primarias, donde el caché del cliente se vuelve una primaria temporal. Para garantizar la consistencia (secuencial), es necesario que se le concedan al cliente permisos exclusivos de escritura, o de lo contrario podrían ocurrir conflictos escritura-escritura.

Los cachés de escritura ofrecen una mejora potencial en el rendimiento con respecto a otros esquemas cuando todas las operaciones pueden realizarse localmente. Podemos hacer mejoras adicionales si retardamos la propagación de las actualizaciones permitiendo que se lleven a cabo escrituras distintas antes de informar a los servidores. Esto da pie a lo que se conoce como **caché de post-escritura**, el cual, de nuevo, se aplica principalmente en sistemas de archivo distribuidos.

7.5.5 Implementación de la consistencia centrada en el cliente

Para nuestro último tema sobre protocolos de consistencia, enfoquémonos en la implementación de la consistencia centrada en el cliente. La implementación de la consistencia centrada en el cliente es relativamente sencilla si ignoramos cuestiones de rendimiento. En las siguientes páginas, primero describiremos dicha implementación, seguida por una descripción de una implementación más realista.

Una implementación simplista

En una implementación simplista de la consistencia centrada en el cliente, a cada operación de escritura *W* se le asigna un identificador global único. Dicho identificador es asignado por el servidor al que se le solicita la escritura. Como en el caso de la consistencia continua, hacemos referencia a este servidor como el origen de *W*. Entonces, para cada cliente, damos seguimiento a dos conjuntos de escrituras. El conjunto de lectura para un cliente consiste en las escrituras importantes para las operaciones de lectura realizadas por el cliente. De manera similar, el conjunto de escrituras consiste en (identificadores de las) escrituras realizadas por el cliente.

La consistencia monotónica de lectura se implementa de la siguiente manera. Cuando un cliente realiza una operación de lectura en un servidor, dicho servidor manipula al conjunto de lecturas del cliente para verificar si todas las escrituras identificadas se realizaron localmente. (El tamaño de dicho conjunto podría provocar un problema de rendimiento para el cual explicaremos una solución más adelante.) Si no, antes de concluir la operación de lectura, hace contacto con los otros servidores para asegurarse de que está al día. De manera alternativa, la operación de lectura se reenvía hacia el servidor en donde se llevaron a cabo las escrituras. Una vez efectuada la operación de lectura, las operaciones de escritura realizadas en el servidor seleccionado, y que son relevantes para la operación de lectura, se adicionan al conjunto de lecturas del cliente.

Observe que sería posible determinar exactamente en dónde se llevaron a cabo las operaciones de escritura identificadas en el conjunto de lectura. Por ejemplo, el identificador de escritura podría incluir el identificador del servidor al que se envió la operación. Se requiere que dicho servidor, por ejemplo, registre la operación de escritura de manera que se pueda repetir en otro servidor. Además, las operaciones de escritura se deben llevar a cabo en el orden en que fueron enviadas. El ordenamiento se puede realizar permitiendo al cliente generar un número de secuencia global y único que se incluya en el identificador de escritura. Si cada elemento de datos sólo puede ser modificado por su propietario, éste puede proporcionar el número de secuencia.

La consistencia monotónica de escritura se implementa de manera análoga a las manufacturas monotónicas. Cada vez que un cliente inicia una nueva operación de escritura en el servidor, a éste se le transfiere el conjunto de escrituras del cliente. (De nuevo, el tamaño del conjunto pudiera ser prohibitivamente grande para los requerimientos de rendimiento. Más adelante explicaremos una solución alternativa.) El servidor garantiza entonces que las operaciones de escritura identificadas se realicen primero y en el orden correcto. Después de realizar la nueva operación, el identificador de dicha operación de escritura se adiciona al conjunto de escritura. Observe que actualizar el servidor con el conjunto de escritura del cliente podría provocar un aumento considerable en el

tiempo de respuesta del cliente, ya que entonces el cliente espera hasta que la operación termina por completo.

De manera similar, la consistencia lea sus escrituras requiere que el servidor donde se realiza la lectura haya visto todas las operaciones de escritura en el conjunto de escritura del cliente. Las escrituras simplemente pueden ser traídas desde otros servidores, antes de que se realice la operación de lectura, aunque esto podría provocar un pobre tiempo de respuesta. De manera alternativa, el software del lado del cliente puede buscar un servidor en donde las operaciones de escritura identificadas en el conjunto de escritura del cliente ya se hayan realizado.

Por último, la consistencia las lecturas siguen a las escrituras puede implementarse al actualizar primero el servidor seleccionado con las operaciones de escritura realizadas en el conjunto de lectura del cliente, y posteriormente agregar el identificador de la operación de escritura al conjunto de escritura, junto con los identificadores en el conjunto de lectura (el cual se ha vuelto relevante para las operaciones de escritura recién realizadas).

Cómo mejorar la eficiencia

Es fácil advertir que el conjunto de lectura y escritura asociado con cada cliente puede volverse muy grande. Para mantener manejables estos conjuntos, las operaciones de lectura y escritura de un cliente se agrupan en sesiones. Por lo general, una **sesión** está asociada con una aplicación: ésta se abre cuando la aplicación comienza, y se cierra cuando finaliza. Sin embargo, las sesiones también se pueden asociar con aplicaciones detenidas de manera temporal, tal como los agentes de usuario en el correo electrónico. Cada vez que un cliente cierra una sesión, los conjuntos simplemente son vaciados. Por supuesto, si un cliente abre una sesión que nunca se cierra, los conjuntos de lectura y escritura asociados pueden volverse muy grandes.

El problema principal con la implementación simplista radica en la representación de los conjuntos de lectura y escritura. Cada conjunto consta de un número de identificadores para operaciones de escritura. Cada vez que un cliente remite una petición de lectura o escritura hacia un servidor, también se transmite al servidor un conjunto de identificadores para ver si todas las operaciones de escritura importantes para la petición se han realizado en dicho servidor.

Esta información puede representarse de manera más eficiente mediante registros de tiempo vectoriales como sigue. Primero, cada vez que un servidor acepta una nueva operación de escritura W, asigna a dicha operación un identificador único global junto con el registro de tiempo $ts(W)$. A una operación de escritura subsiguiente, enviada a dicho servidor, se le asigna un registro de tiempo de más alto valor. Cada servidor S_i mantiene un registro de tiempo vectorial WVC_i, donde $WVC_i[j]$ es igual al registro de tiempo de la operación de escritura más reciente originada a partir de S_j que fue procesada por S_i. Con fines de claridad, supongamos que por cada servidor, las escrituras de S_j se procesan en el orden en que fueron remitidas.

Siempre que un cliente envía una petición para realizar una operación de lectura o escritura O en un servidor específico, dicho servidor devuelve su registro de tiempo actual junto con los resultados de O. Posteriormente, los conjuntos de lectura y escritura se representan mediante registros de tiempo vectoriales. De manera más específica, para cada sesión A, elaboramos un registro de tiempo vectorial SVC_A con $SVC_A[i]$ igual al registro de tiempo máximo de todas las operaciones de escritura en A que se originan desde el servidor S_i:

$$SVC_A[j] = \text{máx}\{\ ts(W) \mid W \in A\ \&\ origen(W) = S_j\ \}$$

En otras palabras, el registro de tiempo de una sesión siempre representa la última operación de escritura que han visto las aplicaciones en ejecución como parte de dicha sesión. La compactación se obtiene mediante la representación de todas las escrituras observadas que se originan desde el mismo servidor a través de un simple registro de tiempo.

Como ejemplo, suponga que un cliente, como parte de una sesión A, se registra en el servidor S_i. Con este fin, pasa SVC_A a S_i. Suponga que $SVC_A[j] > WVC_i[j]$. Lo que esto significa es que S_i no ha visto aún todas las escrituras que se originan en S_j, y que el cliente ya ha visto. De acuerdo con la consistencia requerida, el servidor S_i podría tener que traer estas escrituras antes de poder obtener respuesta para el cliente de manera consistente. Una vez que la operación se realiza, el servidor S_i devuelve el registro de tiempo actual WVC_i. En este punto, SVC_A se ajusta como:

$$SVC_A[j] \leftarrow \text{máx}\ \{\ SVC_A[j],\ WVC_i[j]\}$$

De nuevo, vemos cómo los registros de tiempo vectoriales pueden proporcionar una manera elegante y compacta de representar la historia de un sistema distribuido.

7.6 RESUMEN

Existen primordialmente dos razones para implementar la replicación de datos: mejorar la confiabilidad de un sistema distribuido y mejorar el rendimiento. La replicación introduce un problema de consistencia: cada vez que una réplica se actualiza, se vuelve diferente de las otras réplicas. Para mantener réplicas consistentes, necesitamos propagar las actualizaciones de tal manera que las inconsistencias temporales no se perciban. Por desgracia, hacer eso pudiera degradar severamente el rendimiento, específicamente en sistemas distribuidos a gran escala.

La única solución a este problema es relajar la consistencia de alguna manera. Existen diferentes modelos de consistencia. Para la consistencia continua, la meta es establecer los límites para la desviación numérica entre las réplicas, la desviación de discontinuidad, y las desviaciones en el ordenamiento de las operaciones.

La desviación numérica se refiere al valor por el que las réplicas pueden ser diferentes. Este tipo de desviación es muy dependiente de la aplicación, pero puede, por ejemplo, utilizarse en replicación de acciones. La desviación de discontinuidad se refiere al tiempo por el que una réplica aún se considera consistente, a pesar de que las actualizaciones pudieran haberse realizado tiempo atrás. La desviación de discontinuidad frecuentemente es utilizada por cachés web. Por último, la desviación de ordenamiento se refiere al número máximo de escrituras tentativas que pueden estar pendientes en cualquier servidor sin tener que sincronizarse con los otros servidores de réplicas.

Desde hace mucho, el ordenamiento consistente de operaciones ha conformado la base de muchos modelos de consistencia. Existen muchas variantes, pero parece que sólo algunas prevalecerán entre los desarrolladores de aplicaciones. La consistencia secuencial proporciona básicamente la semántica que el programador espera en la programación concurrente: todas las operaciones son

vistas por todos en el mismo orden. Menos utilizada, pero aún importante, es la consistencia causal, la cual refleja que las operaciones potencialmente dependientes entre sí se realizan en el orden de dicha dependencia.

Modelos de consistencia más débiles consideran series de operaciones de lectura y escritura. En particular, suponen que cada serie es adecuadamente "puesta entre corchetes" por las operaciones que acompañan a las variables de sincronización, en forma de candados. Aunque esto requiere un esfuerzo explícito de los programadores, en general tales modelos son más fáciles de implementar de forma eficiente que, por ejemplo, la consistencia secuencial pura.

Opuestos a estos modelos centrados en los datos, los investigadores que trabajan en el campo de las bases de datos distribuidas para usuarios móviles han definido varios modelos de consistencia centrada en el cliente. Tales modelos no consideran el hecho de que los datos pueden estar compartidos por varios usuarios; en vez de eso, se concentran en la consistencia que se le debe ofrecer a un cliente individual. La suposición subyacente es que un cliente se conecta con diferentes réplicas en el transcurso del tiempo, pero que tales diferencias deben ser transparentes. En esencia, los modelos de consistencia centrada en el cliente garantizan que siempre que un cliente se conecte con una nueva réplica, dicha réplica es actualizada con los datos que han sido manipulados antes por ese cliente, y que probablemente residan en otros sitios de réplica.

Para propagar actualizaciones, es posible aplicar diferentes técnicas. Es necesario hacer una distinción con respecto a *qué* se propaga exactamente, hacia *dónde* se propagan las actualizaciones, y *por quién* se inicia la propagación. Nosotros podemos decidir propagar notificaciones, operaciones, o estados. Asimismo, no toda réplica necesita actualizarse de inmediato. Cuál réplica se actualiza y en qué momento, depende del protocolo de distribución. Por último, se puede elegir si las actualizaciones se insertan en otras réplicas, o qué réplica extrae actualizaciones de otras réplicas.

Los protocolos de consistencia describen implementaciones específicas de modelos de consistencia. Con respecto a la consistencia secuencial y sus variantes, es posible diferenciar a los protocolos basados en primarias y a los protocolos de escrituras replicadas. En los protocolos basados en primarias, todas las operaciones de actualización se remiten a una copia primaria que posteriormente garantiza que la actualización sea ordenada adecuadamente y remitida. En los protocolos de escrituras replicadas, una actualización se remite a varias réplicas al mismo tiempo. En ese caso, el ordenamiento adecuado de las operaciones se vuelve más difícil.

PROBLEMAS

1. El acceso a objetos compartidos de Java puede serializarse al declarar sus métodos como sincronizados. ¿Es suficiente con garantizar la serialización cuando un objeto compartido es replicado?

2. Explique con sus propias palabras cuál es la razón principal para considerar modelos de consistencia débiles.

3. Explique cómo ocurre la replicación en DNS, y por qué en realidad funciona tan bien.

4. Durante la explicación de los modelos de consistencia, a menudo nos referimos al contrato entre el software y el almacén de datos. ¿Por qué es necesario establecer dicho contrato?

5. Dadas las réplicas de la figura 7-2, ¿qué tendríamos que hacer para ultimar los valores de la conit en tal forma que tanto *A* como *B* vieran el mismo resultado?

6. En la figura 7-7, ¿001110 es una salida legal para una memoria secuencialmente consistente? Explique su respuesta.

7. Con frecuencia se dice que los modelos de consistencia débil imponen una carga adicional a los programadores. ¿Hasta qué punto es cierta esta afirmación?

8. La multidifusión totalmente ordenada por medio de un secuenciador, y en favor de la consistencia en la replicación activa, ¿viola el argumento fin a fin del diseño de sistemas?

9. ¿Qué tipo de consistencia utilizaría usted para implementar un mercado electrónico de acciones? Explique su respuesta.

10. Considere un buzón electrónico personal para un usuario móvil, implementado como parte de una base de datos distribuida de área amplia. ¿Qué clase de consistencia centrada en el cliente sería la más adecuada?

11. Describa una implementación sencilla de consistencia lea sus escrituras para desplegar páginas web que acaban de ser actualizadas.

12. Para simplificar las cosas, supusimos que no se presentaban conflictos escritura-escritura en Bayou. Por supuesto, ésta es una suposición nada realista. Explique cómo pueden ocurrir conflictos.

13. Cuando se utiliza un contrato, ¿es necesario que los relojes del cliente y del servidor estén bastante bien sincronizados?

14. Hemos establecido que la multidifusión totalmente ordenada utilizando los relojes lógicos de Lamport no se escala. Explique por qué.

15. Demuestre que, en el caso de la consistencia continua, hacer que el servidor S_k adelante su vista $TW_k(i,k)$ siempre que reciba una actualización reciente que incrementaría $TW(k,k) - TW_k(i,k)$ más allá de $\delta_i/N - 1$, garantiza que $v(t) - v_i \leq \delta_i$.

16. Para la consistencia continua, hemos supuesto que cada escritura sólo incrementa el valor del elemento x. Esquematice una solución en la que también sea posible disminuir el valor de x.

17. Considere un protocolo primario de respaldo de no bloqueo utilizado para garantizar la consistencia secuencial en un almacén de datos distribuido. ¿Dicho almacén de datos siempre proporciona consistencia lea sus escrituras?

18. Para que la replicación activa funcione, en general, es necesario que todas las operaciones se realicen en el mismo orden en cada réplica. ¿Siempre es necesario este ordenamiento?

19. Para implementar la multidifusión totalmente ordenada por medio de un secuenciador, un método es remitir primero una operación al secuenciador, el cual después le asigna un número único y posteriormente multidifunde la operación. Mencione dos métodos alternos, y compare las tres soluciones.

20. Un archivo es replicado en 10 servidores. Haga una lista de todas las combinaciones de quórum de lectura y de quórum de escritura que permite el algoritmo de votación.

21. Los contratos basados en estado se utilizan para descargar un servidor, permitiéndole dar seguimiento a tantos clientes como necesite. ¿Este método conduce necesariamente a un mejor rendimiento?

22. (**Tarea para el laboratorio.**) En este ejercicio, usted implementará un sistema sencillo que soporte la multidifusión (multicast) RPC. Suponemos que hay varios servidores replicados y que cada cliente se comunica con un servidor mediante RPC. Sin embargo, cuando se trata con la replicación, un cliente necesitará enviar una petición RPC a cada réplica. Programe al cliente en tal forma que para la aplicación parezca como si sólo se enviara un RPC. Suponga que está replicando por rendimiento, pero que los servidores son susceptibles a fallas.

8

TOLERANCIA A FALLAS

Una característica sobresaliente de los sistemas distribuidos que los distingue de los sistemas de una sola máquina es la noción de falla parcial. En un sistema distribuido, una falla parcial puede acontecer cuando falla un componente. Esta falla puede afectar la operación de algunos componentes, al tiempo que otros más no se ven afectados en absoluto. Por contraste, en un sistema no distribuido, una falla a menudo es total en el sentido de que afecta a todos los componentes y fácilmente puede echar abajo al sistema.

Un objetivo importante en el diseño de sistemas distribuidos es construirlos de manera que puedan recuperarse automáticamente de fallas parciales sin que se afecte seriamente el desempeño total. En particular, siempre que ocurra una falla, el sistema distribuido deberá continuar operando de modo aceptable mientras se realizan reparaciones, es decir, deberá tolerar las fallas y continuar operando hasta cierto grado incluso en su presencia.

En este capítulo, examinamos más a fondo las técnicas apropiadas para hacer que los sistemas distribuidos toleren las fallas. Después de proporcionar algunas bases generales sobre tolerancia a las fallas, daremos un vistazo a la atenuación del proceso y a la multitransmisión confiable. La atenuación del proceso incorpora técnicas mediante las cuales uno o más procesos pueden fallar sin perturbar seriamente el resto del sistema. Relacionada con este tema está la multitransmisión, gracias a la cual se garantiza la transmisión exitosa de un mensaje hacia un conjunto de procesos. La multitransmisión confiable a menudo es necesaria para mantener sincronizado el proceso.

La atomicidad es una propiedad importante en muchas aplicaciones. Por ejemplo, en transacciones distribuidas, es necesario garantizar que se realicen todas las operaciones involucradas en un proceso o que no se realice ninguna. La noción de protocolos de dedicación es fundamental

para implementar la atomicidad en los sistemas distribuidos, y se presenta en otra sección de este capítulo.

Por último, examinaremos cómo recuperarse de una falla. En particular, cuándo y cómo se deberá guardar el estado de un sistema distribuido para recuperarlo más adelante.

8.1 INTRODUCCIÓN A LA TOLERANCIA A LAS FALLAS

La tolerancia a las fallas se ha investigado mucho en la ciencia de la computación. En esta sección, primero se presentan los conceptos básicos relacionados con las fallas de procesamiento, seguidos por un análisis de modelos de fallas. La técnica clave para el manejo de fallas es la redundancia, que también estudiamos. Para información más general sobre tolerancia a las fallas en sistemas distribuidos vea, por ejemplo, Jalote (1994) o (Shooman, 2002).

8.1.1 Conceptos básicos

Para entender el rol de la tolerancia a fallas en los sistemas distribuidos, primero se tiene que examinar a fondo lo que en realidad significa el que un sistema distribuido tolere fallas. Ser tolerante a las fallas está fuertemente relacionado con lo que se llama **sistemas fiables**. Fiabilidad es un término que comprende varios requerimientos útiles para los sistemas distribuidos incluidos los siguientes (Kopetz y Verissimo, 1993):

1. Disponibilidad

2. Confiabilidad

3. Seguridad

4. Mantenimiento

Disponibilidad se define como la propiedad de que un sistema está listo para ser utilizado de inmediato. En general, se refiere a la probabilidad de que el sistema esté operando correctamente en cualquier momento dado y se encuentre disponible para realizar sus funciones a nombre de sus usuarios. En otros términos, un sistema altamente disponible es uno que muy probablemente funcionará en un instante dado.

Confiabilidad se refiere a la propiedad de que un sistema sea capaz de funcionar de manera continua sin fallar. Por contraste con la disponibilidad, la confiabilidad se define en función de un intervalo de tiempo en lugar de un instante en el tiempo. Un sistema altamente confiable es uno que muy probablemente continuará funcionando sin interrupción durante un lapso de tiempo relativamente largo. Ésta es una sutil pero importante diferencia cuando se compara con la disponibilidad. Si un sistema se viene abajo durante un milisegundo por hora, su disponibilidad es de más del

99.9999 por ciento, pero no es confiable. Asimismo, un sistema que nunca se congela pero que deja de funcionar dos semanas cada agosto es altamente confiable, aunque sólo esté un 96% disponible. Estas dos situaciones no son lo mismo.

Seguridad se refiere a la situación en que no acontece nada catastrófico cuando un sistema deja de funcionar correctamente durante un tiempo. Por ejemplo, se requiere que muchos sistemas de control de proceso, como los utilizados para controlar plantas de energía nuclear o enviar personas al espacio exterior, proporcionen un alto grado de seguridad. Si tales sistemas de control fallan temporalmente durante sólo un breve momento, los efectos podrían ser desastrosos. Muchos ejemplos del pasado (y probablemente más que aún no acontecen) demuestran lo difícil que es construir sistemas seguros.

Por último, el **mantenimiento** se refiere a cuán fácil puede ser reparado un sistema que falló. Un sistema altamente mantenible también puede ser altamente disponible, en especial si las fallas pueden ser detectadas y reparadas en forma automática. Sin embargo, como se verá más adelante en este capítulo, la recuperación automática de fallas es fácil de expresar pero difícil de realizar.

A menudo, también se requiere que los sistemas fiables proporcionen un alto grado de seguridad, en especial cuando se trata de temas tales como la integridad. La seguridad se analizará en el siguiente capítulo.

Se dice que un sistema **falla** cuando no puede cumplir sus promesas. En particular, si un sistema distribuido se diseña para proporcionar a sus usuarios varios servicios, el sistema falla cuando uno o más de esos servicios no puede ser proporcionado (a cabalidad). Un **error** es una parte del estado de un sistema que puede conducir a una falla. Por ejemplo, cuando se transmiten paquetes a través de una red, es de esperarse que algunos le lleguen dañados al destinatario. Dañado en este contexto significa que el destinatario puede detectar incorrectamente un valor de bit (por ejemplo, leer un 1 en lugar de un 0), o incluso puede ser incapaz de detectar que llegó algo.

La causa de un error se llama **falla**. Claramente, indagar la causa del error es importante. Por ejemplo, un medio de transmisión incorrecto o dañado puede fácilmente dañar los paquetes. En este caso, es relativamente sencillo eliminar la falla. Sin embargo, las condiciones climáticas también pueden provocar errores de transmisión como, por ejemplo, en redes inalámbricas. Cambiar el clima para reducir o evitar errores es más que difícil.

La construcción de sistemas fiables tiene mucho que ver con el control de fallas. Se puede hacer una distinción entre evitar, eliminar, y pronosticar fallas (Avizienis y cols., 2004). Para nuestros propósitos, el tema más importante es la **tolerancia a las fallas** lo que significa que un sistema puede proveer sus servicios incluso en presencia de fallas. En otros términos, el sistema puede tolerarlas y continuar operando con normalidad.

Las fallas se clasifican generalmente como transitorias, intermitentes, o permanentes. Las **fallas transitorias** ocurren una vez y luego desaparecen. Si la operación se repite la falla desaparece. Un pájaro que vuele a través del haz de la señal de un transmisor de microondas puede hacer que se pierdan bits en alguna red (sin mencionar un pájaro rostizado). Si la transmisión se interrumpe y se intenta restaurarla, probablemente funcionará la segunda vez.

Una **falla intermitente** ocurre, luego desaparece por sí sola, después reaparece, y así sucesivamente. Un falso contacto en un conector a menudo será la causa de una falla intermitente. Las fallas intermitentes provocan una gran cantidad de problemas porque son difíciles de diagnosticar. Típicamente, cuando el doctor de fallas hace acto de presencia, el sistema funciona bien.

Una **falla permanente** es una que continúa existiendo hasta que el componente defectuoso es reemplazado. Chips fundidos, errores en programas, y roturas de la cabeza de lectura-escritura de un disco son ejemplos de fallas permanentes.

8.1.2 Modelos de falla

Un sistema que falla no proporciona adecuadamente los servicios para los que fue diseñado. Si se considera un sistema distribuido como un conjunto de servidores que se comunican entre sí y con sus clientes, no proporcionar adecuadamente los servicios significa que los servidores, los canales de comunicación, o posiblemente ambos, no están haciendo lo que se supone deben hacer. Sin embargo, un servidor que funciona mal no siempre es la falla buscada. Si ese servidor depende de otros servidores para proporcionar adecuadamente sus servicios, la causa del error tiene que buscarse en otra parte.

Tales relaciones de dependencia aparecen en abundancia en sistemas distribuidos. Un disco que falla puede complicarle la vida a un servidor de archivos diseñado para proporcionar un sistema de archivos altamente disponible. Si tal servidor de archivos forma parte de una base de datos distribuida, el funcionamiento apropiado de toda la base de datos puede estar en peligro ya que sólo una parte de sus datos puede ser accesible.

Para tener una mejor idea de qué tan seria es en realidad una falla, se han desarrollado varios esquemas de clasificación. Uno de éstos se muestra en la figura 8-1, y está basado en esquemas descritos en Cristian (1991) y Hadzilacos y Toueg (1993).

Tipo de falla	Descripción
Falla de congelación	Un servidor se detiene, pero estaba trabajando correctamente hasta que se detuvo
Falla de omisión	Un servidor no responde a las peticiones entrantes
Omisión de recepción	Un servidor no recibe los mensajes entrantes
Omisión de envío	Un servidor no envía mensajes
Falla de tiempo	La respuesta de un servidor queda fuera del intervalo de tiempo especificado
Falla de respuesta	La respuesta de un servidor es incorrecta
Falla de valor	El valor de la respuesta está equivocado
Falla de transición de estado	El servidor se desvía del flujo de control correcto
Falla arbitraria	Un servidor puede producir respuestas arbitrarias en tiempos arbitrarios

Figura 8-1. Diferentes tipos de fallas.

Una **falla de congelación** ocurre cuando un servidor se detiene prematuramente, pero el cual estaba funcionando correctamente hasta entonces. Un aspecto importante de las fallas de congelación es que una vez detenido el servidor, ya no se sabe nada de él. Un ejemplo típico de una falla

de congelación es un sistema operativo que se detiene de repente, y para lo cual sólo hay una solución: reiniciarlo. Muchos sistemas de computadora personal experimentan fallas de congelación con tanta frecuencia que todo mundo lo considera normal. Por consiguiente, el cambio del botón de reinicio de la parte posterior del gabinete a la parte frontal se hizo por una buena razón. Quizás un día pueda volverse a colocar detrás o incluso eliminarlo del todo.

Una **falla de omisión** ocurre cuando un servidor no atiende una petición. Varias cosas pueden estar mal. En el caso de una falla de omisión, en primer lugar, posiblemente el servidor nunca obtuvo la petición. Observemos que muy bien puede ser el caso de que la conexión entre un cliente y un servidor haya sido establecida correctamente, pero que no se escucharon las peticiones entrantes. También, una falla de omisión de recepción en general no afecta el estado actual del servidor, ya que éste ignora que le enviaron un mensaje.

Asimismo, se presenta una falla de omisión de envío cuando el servidor ha hecho su trabajo, pero algo salió mal al enviar una respuesta. Esa falla puede suceder, por ejemplo, cuando un bufer de envío se sobresatura y el servidor no estaba preparado para enfrentar esa situación. Observemos que, por contraste con una falla de omisión de recepción, el servidor ahora puede encontrarse en un estado que refleja que acaba de completar un servicio para el cliente. En consecuencia, si el envío de su respuesta falla, el servidor debe estar preparado para que el cliente reenvíe su solicitud previa.

Otros tipos de fallas de omisión que no están relacionados con la comunicación pueden ser provocados por errores de software tales como bucles infinitos o una gestión inadecuada de la memoria por la cual se dice que el servidor “se colgó”.

Otra clase de fallas tiene que ver con la temporización. Las **fallas de temporización** ocurren cuando la respuesta queda fuera de un intervalo de tiempo real especificado. Como se vio en el capítulo 4 con los flujos de datos isócronos, proporcionar datos demasiado pronto fácilmente puede provocar problemas a un destinatario si no hay suficiente espacio en el búfer para guardar todos los datos entrantes. Más común, sin embargo, es que un servidor responda demasiado tarde, en cuyo caso se dice que ocurre una falla de *desempeño*.

Un tipo serio de falla es una **falla de respuesta**, por la cual la respuesta del servidor es simplemente incorrecta. Pueden suceder dos clases de fallas de respuesta. En el caso de una falla de valor, un servidor simplemente responde en forma equivocada a una petición. Por ejemplo, un motor de búsqueda que sistemáticamente regresa páginas web que no están relacionadas con cualesquiera de los términos de búsqueda usados, ha fallado.

El otro tipo de falla de respuesta se conoce como **falla de transición de estado**. Esta clase de falla ocurre cuando el servidor reacciona inesperadamente a una solicitud entrante. Por ejemplo, si un servidor recibe un mensaje que no puede reconocer, ocurrirá una falla de transición si no se tomaron las medidas apropiadas para manejar tales mensajes. En particular, un servidor defectuoso puede tomar incorrectamente medidas preestablecidas que nunca debería haber iniciado.

Las más serias son **fallas arbitrarias**, también conocidas como **fallas bizantinas**. En realidad, cuando ocurren fallas arbitrarias, los clientes deberían estar preparados para lo peor. En particular, suele suceder que un servidor produzca una salida que nunca debió haber producido, pero la cual no puede ser detectada como incorrecta. Peor todavía, un servidor defectuoso incluso puede estar trabajando maliciosamente con otros servidores para producir de manera intencional respuestas erróneas. Esta situación ilustra porqué la seguridad también se considera

un requerimiento importante cuando se habla de sistemas fiables. El término "bizantino" se refiere al Imperio Bizantino, una época (330-1453) y un lugar (los Balcanes y la moderna Turquía) donde las conspiraciones interminables, las intrigas, y la desconfianza eran cosa de todos los días en los círculos gubernamentales. Las fallas bizantinas fueron analizadas primero por Pease y colaboradores (1980) y Lamport y colaboradores (1982). Más adelante regresaremos al tema.

Las fallas arbitrarias están estrechamente relacionadas con las fallas de congelación. La definición de fallas de congelación tal como se presentó con anterioridad es la forma más benigna de detención de un servidor. También se conocen como **fallas de detención**. En efecto, un servidor que sufre este tipo de falla simplemente deja de producir una salida de tal forma que su detención puede ser detectada por otros procesos. En el mejor de los casos, el servidor puede haber sido tan amistoso de avisar que estaba a punto de congelarse: de lo contrario simplemente se detiene.

Desde luego, en la vida real, los servidores se detienen por omisión o congelación y no avisan de antemano que se van a detener. Les toca a otros procesos decidir que un servidor se ha detenido prematuramente. Sin embargo, en tales **sistemas silenciosos**, el servidor puede volverse inesperadamente lento, es decir, exhibir fallas de desempeño.

Por último, también hay ocasiones en las cuales el servidor produce una salida aleatoria, aunque esta salida puede ser reconocida por otros procesos como simple basura. El servidor exhibe entonces fallas arbitrarias, pero de manera benigna. Estas fallas también se conocen como **fallas seguras**.

8.1.3 Disfrazado de fallas por redundancia

Para que un sistema sea tolerante a fallas, lo mejor que se puede hacer es tratar de ocultar ante otros procesos la ocurrencia de fallas. La técnica clave para disfrazar las fallas es utilizar redundancia. Tres clases son posibles: redundancia de información, redundancia de tiempo, y redundancia física [vea también Johnson (1995)]. Con redundancia de información, se agregan bits adicionales para recuperar los bits mutilados. Por ejemplo, se puede agregar un código Hamming a los datos transmitidos para recuperarlos del ruido presente en la línea de transmisión.

Con redundancia de tiempo, se realiza una acción, y luego, si es necesario, se vuelve a realizar. Las transacciones (vea el capítulo 1) utilizan este método. Si se aborta una transacción, puede rehacerse sin ningún perjuicio. La redundancia de tiempo resulta especialmente útil cuando las fallas son transitorias e intermitentes.

Con redundancia física, se agrega equipo o procesos adicionales para que el sistema en su conjunto tolere la pérdida o el mal funcionamiento de algunos componentes. La redundancia física puede realizarse entonces en el hardware o el software. Por ejemplo, se pueden agregar procesos adicionales al sistema de modo que si algunos se congelan, el sistema pueda seguir funcionando correctamente. En otros términos, al replicar procesos se logra un alto grado de tolerancia a fallas. Más adelante volveremos a tratar este tipo de redundancia de software.

La redundancia física es una técnica muy conocida de crear tolerancia a fallas. Se utiliza en biología (los mamíferos tienen dos ojos, dos oídos, dos pulmones, etc.), aeronáutica (los

aviones 747 tienen cuatro motores pero pueden volar con tres), y deportes (varios árbitros por si a uno se le pasa un suceso). También se ha utilizado tolerancia a fallas en circuitos electrónicos durante años: resulta muy ilustrativo analizar cómo se ha aplicado en ese campo. Consideremos, por ejemplo, el circuito de la figura 8-2(a). Las señales pasan a través de los dispositivos *A*, *B* y *C*, en secuencia. Si un dispositivo falla, el resultado final probablemente será incorrecto.

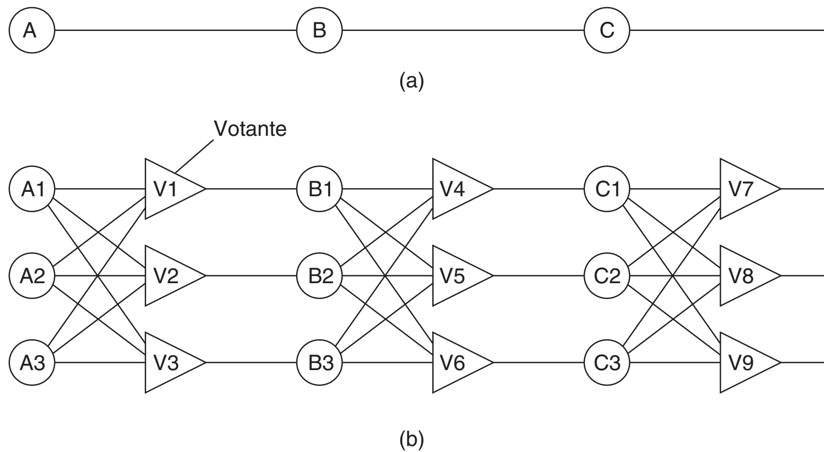


Figura 8-2. Redundancia modular triple.

En la figura 8-2(b), cada dispositivo está replicado tres veces. Después de cada etapa del circuito hay un votante triplicado. Cada votante es un circuito que tiene tres entradas y una salida. Si dos o tres de las entradas son las mismas, la salida es igual a esa entrada. Si las tres entradas son diferentes, la salida es indefinida. Esta clase de diseño se conoce como **TMR** (*Triple Modular Redundancy*; **redundancia modular triple**).

Supongamos que falla el elemento A_2. Cada uno de los votantes V_1, V_2 y V_3 obtiene dos entradas buenas (idénticas) y una maliciosa, y cada entrada transfiere el valor correcto a la segunda etapa. En esencia, el efecto de la falla de A_2 es disfrazado por completo, de modo que las entradas a B_1, B_2 y B_3 sean exactamente las mismas como si no hubiera ocurrido ninguna falla.

Ahora consideremos lo que sucede si B_3 y C_1 también fallan, además de A_2. Estos efectos también son disfrazados, de modo que las tres salidas finales sigan siendo correctas.

Al principio puede no ser evidente por qué se requieren tres votantes en cada etapa. Después de todo, un votante podría también detectar y pasar la inspección de la mayoría. Sin embargo, un votante es además un componente y también puede fallar. Supongamos, por ejemplo, que el votante V_1 funciona mal. La entrada a B_1 estará entonces equivocada, pero en tanto todo lo demás funcione, B_2 y B_3 producirán la misma salida y V_4, V_5 y V_6 enviarán el resultado correcto a la etapa tres. Una falla en V_1 efectivamente no es diferente de una falla en B_1. En ambos casos B_1

produce una salida incorrecta, pero en ambos casos es rechazada y el resultado final seguirá siendo correcto.

Aunque no todos los sistemas distribuidos tolerantes a las fallas utilizan TMR, la técnica es muy general, y deberá darnos una buena idea de lo que es un sistema tolerante a fallas, contrario a un sistema cuyos componentes individuales son altamente confiables pero cuya organización no puede tolerar fallas (es decir, operar correctamente incluso en presencia de componentes defectuosos). Por supuesto, la redundancia modular triple puede ser aplicada repetidamente, por ejemplo, para hacer que un chip sea altamente confiable con redundancia modular triple en su interior, sin que los diseñadores que lo utilizan lo sepan, posiblemente en su propio circuito que contiene múltiples copias de los chips junto con votantes.

8.2 ATENUACIÓN DE UN PROCESO

Ahora que ya se tocaron los temas básicos de tolerancia a fallas, veremos cómo se puede lograr en realidad la tolerancia a fallas en sistemas distribuidos. El primer tema a abordar es la protección contra fallas de proceso, la cual se logra replicando los procesos en grupos. En las páginas siguientes, consideramos temas de diseño generales de grupos de procesos y analizamos lo que es realmente un grupo tolerante a fallas. También, vemos cómo llegar a un acuerdo dentro de un grupo de procesos cuando no se confía en que uno o más de sus miembros den respuestas correctas.

8.2.1 Temas de diseño

La forma clave de afrontar la tolerancia a un proceso defectuoso es organizar varios procesos idénticos en un grupo. La propiedad fundamental que tienen todos los grupos es que cuando un mensaje es enviado al grupo, todos los miembros de éste lo reciben. De esta manera, si en un grupo falla un proceso, afortunadamente alguno de los demás procesos puede hacerse cargo de él (Guerraoui y Schiper, 1997).

Los grupos de procesos pueden ser dinámicos. Se pueden crear grupos nuevos y destruir los viejos. Un proceso puede unirse a un grupo o abandonarlo durante la operación del sistema. Un proceso puede ser miembro de varios grupos a la vez. En consecuencia, se requieren mecanismos adecuados para gestionar los grupos y la membresía a un grupo.

Los grupos son más o menos como las organizaciones sociales. Alicia podría ser miembro de un club de lectura, un club de tenis, y una organización ecológica. En un día particular, podría recibir correos (mensajes) para informarle sobre un nuevo libro de repostería del club de lectura, de un torneo de tenis por el día de las madres del club de tenis, y del inicio de una campaña para salvar la marmota del sur de la organización ecológica. En cualquier momento, Alicia tiene la libertad de dejar cualquiera o todos estos grupos y posiblemente unirse a otros.

El propósito de la introducción de grupos es permitir que los procesos se ocupen de los conjuntos de procesos como una sola abstracción. Por tanto, un proceso puede enviar un mensaje a un grupo de servidores sin que deba saber cuáles son o cuántos son o dónde están, lo cual puede cambiar de una invocación a la siguiente.

Grupos de 4 compañeros contra grupos jerárquicos

Una importante distinción entre los diferentes grupos tiene que ver con su estructura interna. En algunos, todos los procesos son iguales; ninguno es el que manda y todas las decisiones se toman colectivamente. En otros grupos, existe alguna clase de jerarquía. Por ejemplo, un proceso es el coordinador y los demás son trabajadores. En este modelo, cuando un cliente externo o uno de los trabajadores genera una solicitud de trabajo, ésta se envía al coordinador. El coordinador decide entonces cuál de los trabajadores es el más apto para realizar el trabajo y lo remite allí. También son posibles jerarquías más complejas, desde luego. Estos patrones de comunicación se ilustran en la figura 8-3.

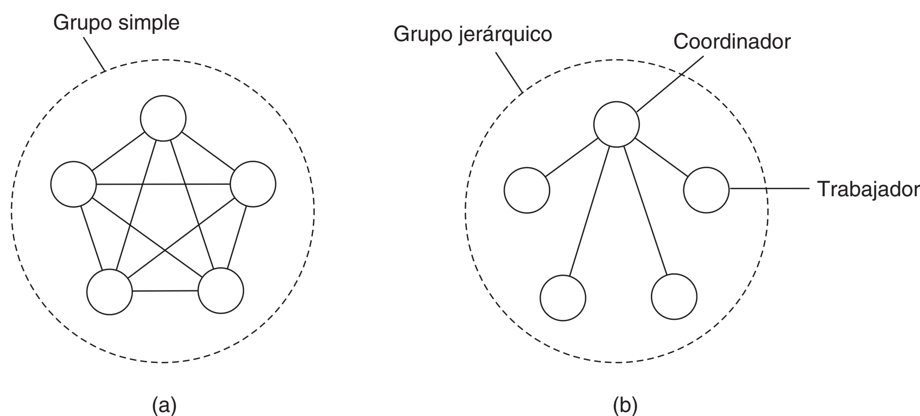


Figura 8-3. (a) Comunicación en un grupo simple. (b) Comunicación en un grupo jerárquico simple.

Cada una de estas organizaciones tiene sus propias ventajas y desventajas. El grupo de compañeros es simétrico y no tiene un solo punto de falla. Si uno de los procesos se congela, el grupo simplemente se vuelve más pequeño, pero por lo demás puede continuar. Una desventaja es que la toma de decisiones resulta más complicada. Por ejemplo, para decidir cualquier cosa, a menudo hay que someterlo a votación, ello implica retraso y gastos indirectos.

El grupo jerárquico tiene las propiedades opuestas. La pérdida del coordinador hace que todo el grupo se detenga bruscamente, pero mientras está funcionando puede tomar decisiones sin molestar a nadie más.

Membresía a un grupo

Cuando en un grupo existe comunicación, se requiere de algún método para crear y eliminar grupos, así como también para permitir que los procesos se unan a los grupos o los abandonen. Un posible método es tener un **servidor de grupo** al cual todas estas peticiones puedan ser enviadas. El servidor de grupo puede mantener entonces una base de datos completa para todos los grupos y su

membresía exacta. Este método es simple, eficiente, y bastante fácil de implementar. Por desgracia, comparte una desventaja importante con todas las técnicas centralizadas: un solo punto de falla. Si el servidor de grupo se congela, la gestión del grupo deja de existir. Probablemente la mayoría o todos los grupos tendrán que ser reconstruidos desde cero, terminando quizá con cualquier trabajo que se estuviera realizando.

El método opuesto es gestionar la membresía de una forma distribuida. Por ejemplo, si la multitransmisión (confiable) está disponible, un extraño puede enviar un mensaje a todos los miembros del grupo para comunicarles su deseo de unirse al grupo.

De modo ideal, para dejar un grupo, un miembro simplemente envía un mensaje de despedida a todos. En el contexto de tolerancia a fallas, la suposición de la semántica de falla de detención generalmente no es apropiada. El problema es que no existe un aviso cortés de que el proceso se congeló como lo hay cuando un proceso abandona voluntariamente. Los demás miembros tienen que descubrir esto en forma experimental cuando se dan cuenta de que el miembro congelado ya no responde a nada. Una vez seguros de que el miembro realmente está congelado (y no sólo lento), podemos eliminarlo del grupo.

Otro tema intrincado es que el abandono y la unión tienen que sincronizarse con mensajes de datos que se están enviando. En otros términos, a partir del instante en que un proceso se une a un grupo, deberá recibir todos los mensajes enviados al grupo. Asimismo, en cuanto un proceso deja el grupo, ya no debe recibir más mensajes de éste y los demás miembros no deben recibir más mensajes de él. Una forma de asegurarse de que una unión o un abandono están integrados al flujo de mensajes en el lugar apropiado es convertir esta operación en una secuencia de mensajes enviados a todo el grupo.

Un tema final relacionado con la membresía a un grupo es qué hacer cuando muchas máquinas se apagan de tal forma que el grupo ya no puede funcionar. Se requiere de un protocolo para reconstruirlo. Invariablemente, algún proceso tendrá que tomar la iniciativa para poner la bola en juego, pero ¿qué sucede si dos o tres procesos tratan de hacerlo al mismo tiempo? El protocolo debe ser capaz de soportarlo.

8.2.2 Enmascaramiento de fallas y replicación

Los grupos de procesos son parte de la solución implementada para construir sistemas tolerantes a fallas. En particular, tener un grupo de procesos idénticos permite disfrazar uno o más procesos defectuosos presentes en dicho grupo. En otros términos, podemos replicar los procesos y organizarlos en un grupo para reemplazar un solo proceso (vulnerable) con un grupo de procesos (tolerante a fallas). Como se vio en el capítulo anterior, existen dos formas de ocuparse de la replicación: por medio de protocolos basados en un protocolo primario, o mediante protocolos de escritura replicados.

La replicación basada en un protocolo primario, en el caso de tolerancia a fallas, generalmente aparece en la forma de un protocolo de respaldo primario. Así, un grupo de procesos se organiza en una forma jerárquica en la cual un protocolo primario coordina todas las operaciones de escritura. En la práctica, el primario está fijo, aunque su función puede ser tomada por uno de los respaldos, si es necesario. En realidad, cuando el primario se congela, los respaldos ejecutan algún algoritmo de su elección para seleccionar un nuevo primario.

Como explicamos en el capítulo anterior, se utilizan protocolos de escritura replicados en la forma de replicación activa, así como también por medio de protocolos basados en quórum. Estas soluciones corresponden a organizar un conjunto de procesos idénticos en un grupo simple. La ventaja principal es que dichos grupos no tienen un solo punto de falla, a causa de la coordinación distribuida.

Un tema importante con la utilización de grupos de procesos para tolerar fallas es cuánta replicación se requiere. Para simplificar el análisis, consideraremos solamente los sistemas de escritura replicados. Se dice que un sistema **es tolerante a k fallas** si puede sobrevivir a las fallas de k componentes y continuar satisfaciendo sus especificaciones. Si los componentes, por ejemplo procesos, fallan silenciosamente, es suficiente con tener $k + 1$ de ellos para proveer la tolerancia k a fallas. Si k de los componentes simplemente se detienen, podemos utilizar la respuesta del otro componente.

Por otra parte, si los procesos exhiben fallas bizantinas, y siguen funcionando y enviando respuestas erróneas o aleatorias, se requiere un mínimo de $2k + 1$ procesadores para lograr la tolerancia a k fallas. En el peor de los casos, los k procesos que fallan accidentalmente (o a la larga intencionalmente) podrían generar la misma respuesta. No obstante, los $k + 1$ procesos restantes también darán la misma respuesta, así que el cliente o el votante sólo pueden creerle a la mayoría.

Desde luego, en teoría es bueno decir que un sistema es tolerante a k fallas y dejar que las $k + 1$ respuestas idénticas voten contra las k respuestas idénticas, pero en la práctica resulta difícil imaginar circunstancias en las que se pueda afirmar con certeza que k procesos pueden fallar pero $k + 1$ no. Entonces, incluso en un sistema tolerante a fallas puede que se requiera alguna clase de análisis estadístico.

Una condición previa implícita pertinente para este modelo es que todas las peticiones lleguen a todos los servidores en el mismo orden, también se le llama **problema de multitransmisión atómica**. En realidad, esta condición puede mitigarse un poco, puesto que las operaciones de lectura no importan y algunas operaciones de escritura pueden conmutarse, pero el problema general permanece. La multitransmisión atómica se aborda con todo detalle en una sección posterior.

8.2.3 Acuerdo en sistemas defectuosos

La organización de procesos replicados en un grupo incrementa la tolerancia a fallas. Como ya mencionamos, si un cliente puede basar sus decisiones en un mecanismo de votación, incluso puede tolerarse que k de $2k + 1$ procesos mientan sobre su resultado. La suposición que se está haciendo, sin embargo, es que los procesos no hacen equipo para producir un resultado equivocado.

En general, las cosas se complican cuando se demanda que un grupo de procesos llegue a un acuerdo, lo cual se requiere en muchos casos. Algunos ejemplos son: la elección de un coordinador, la decisión de realizar o no una transacción, la división de tareas entre los trabajadores, y la sincronización, entre muchas otras posibilidades. Cuando todos los procesos y la comunicación son perfectos, llegar a semejante acuerdo a menudo es simple, pero cuando no lo son, surgen problemas.

El objetivo general de los algoritmos de acuerdo distribuidos es hacer que todos los procesos no defectuosos alcancen un consenso en algún tema, y establecer dicho consenso dentro de un número finito de pasos. El problema se complica por el hecho de que diferentes suposiciones sobre el sistema subyacente requieren diferentes soluciones, suponiendo incluso que existen soluciones. Turek y Shasha (1992) distinguen los siguientes casos.

1. Sistemas síncronos contra sistemas asíncronos. Un sistema es **síncrono** si, y sólo si, se sabe que los procesos operan por pasos. Formalmente, esto significa que deberá haber una constante $c \geq 1$, de modo que si cualquier proceso tomó $c + 1$ pasos, todos los otros procesos habrán tomado por lo menos 1 paso. Se dice que un sistema no síncrono es **asíncrono**.

2. El retraso de la comunicación está o no limitado. El retraso está limitado si, y sólo si, se sabe que cada mensaje es entregado con un tiempo globalmente máximo y predeterminado.

3. La entrega de mensajes se hace o no en orden. En otros términos, es posible distinguir una situación en la que los mensajes del mismo remitente se entregan en el orden en que fueron enviados de una situación en que no se tienen tales garantías.

4. Los mensajes se transmiten mediante unitransmisión o multitransmisión.

Como resulta ser, llegar a un acuerdo sólo es posible en la situación mostrada en la figura 8-4. En todos los demás casos, se puede demostrar que no existe ninguna solución. Observemos que la mayoría de los sistemas distribuidos presuponen que los procesos se comportan asíncronamente, que los mensajes se transmiten por unitransmisión, y que los retrasos de comunicación no están limitados. En consecuencia, se tiene que utilizar la entrega de mensajes ordenada (confiable), tal como es provista mediante TCP. La figura 8-4 ilustra la naturaleza no trivial del acuerdo distribuido cuando los procesos pueden fallar.

El problema originalmente fue estudiado por Lamport y colaboradores (1982) y también se conoce como **problema de acuerdo bizantino**, ya que hace referencia a numerosas guerras en las que varios ejércitos tuvieron que llegar a un acuerdo, por ejemplo, refuerzo de las tropas mientras se enfrentaban con generales traidores, lugartenientes confabuladores, y así por el estilo. Consideremos la siguiente solución, descrita en Lamport y colaboradores (1982). En este caso, asumimos que los procesos son síncronos, que los mensajes se envían mediante unitransmisión al mismo tiempo que se conserva el orden, y que se delimita el retraso de la comunicación. Se supone que existen N procesos, donde cada proceso i proporcionará un valor v_i a los demás. El objetivo es dejar que cada proceso construya un vector V de longitud N, de modo que si el proceso i no está defectuoso, $V[i] = v_i$. De lo contrario, $V[i]$ está indefinida. Suponemos que existen cuando mucho k procesos defectuosos.

En la figura 8-5 se ilustra el funcionamiento del algoritmo para el caso de $N = 4$ y $k = 1$. Con estos parámetros, el algoritmo opera en cuatro pasos. En el paso 1, cada proceso i no defectuoso envía v_i a todos los demás procesos mediante unitransmisión confiable. Los procesos defectuosos

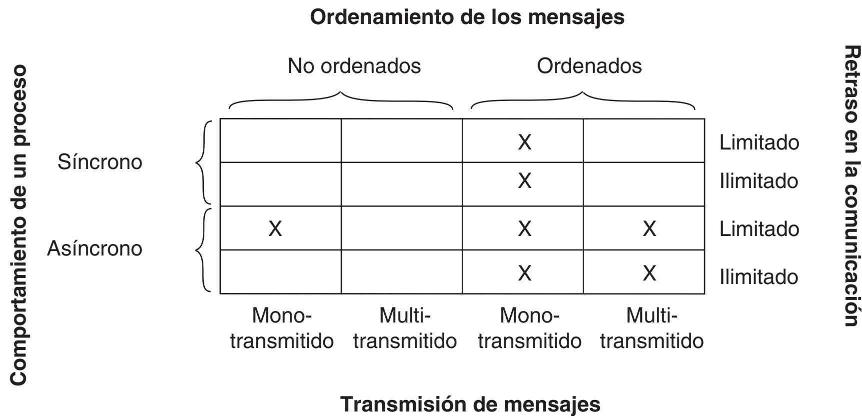


Figura 8-4. Circunstancias en las que el acuerdo distribuido puede ser alcanzado.

pueden enviar cualquier cosa. Además, como estamos utilizando multitranmisión, pueden enviar diferentes valores a diferentes procesos. Sea $v_i = i$. En la figura 8-5(a) se aprecia que el proceso 1 reporta 1, el 2 reporta 2, el 3 les miente a todos, les envía *x*, *y* y *z*, respectivamente, y el proceso 4 reporta un valor de 4. En el paso 2, los resultados de los anuncios del paso 1 se reúnen en la forma de los vectores de la figura 8-5(b).

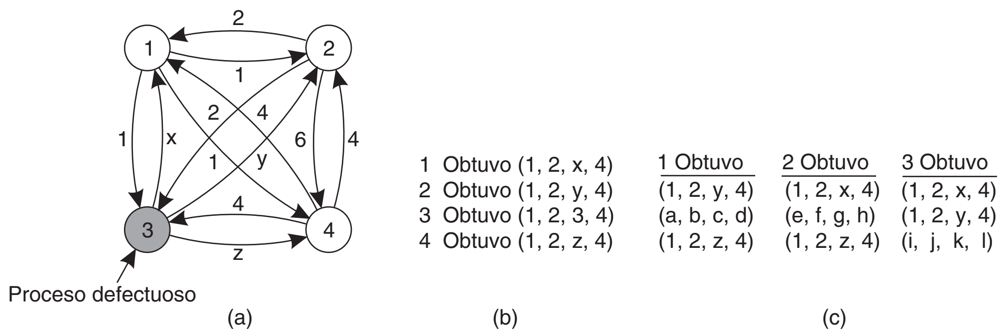


Figura 8-5. Problema del acuerdo bizantino de tres procesos no defectuosos y uno defectuoso. (a) Cada proceso envía su valor a los demás. (b) Los vectores que cada proceso ensambla basado en (a). (c) Los vectores que cada proceso recibe en el paso 3.

En el paso 3 cada proceso transfiere su vector desde la figura 8-5(b) a todos los demás procesos. De este modo, cada proceso obtiene tres vectores, uno de cada proceso. En este caso, también, el proceso 3 miente e inventa 12 nuevos valores, *a* a *l*. Los resultados del paso 3 aparecen en la figura 8-5(c). Finalmente, en el paso 4, cada proceso examina el elemento *i*-ésimo de cada uno de los

vectores recién recibidos. Si cualquier valor tiene mayoría, dicho valor se pone en el vector resultado. Si ningún valor tiene mayoría, el elemento correspondiente del vector resultado se marca mediante *UNKNOWN*. En la figura 8-5(c) vemos que 1, 2, y 4 concordaron en los valores de v_1, v_2 y v_4, que es el resultado correcto. Lo que estos procesos concluyen con respecto a v_3 no puede ser decidido, aunque tampoco viene al caso. El objetivo del acuerdo bizantino es que se llegue a un consenso en cuanto al valor de únicamente los procesos no defectuosos.

Ahora regresaremos a este problema con $N = 3$ y $k = 1$, es decir, sólo dos procesos no defectuosos y uno si, como se ilustra en la figura 8-6. En este caso vemos que en la figura 8-6(c) ninguno de los procesos que se están comportando correctamente ve una mayoría para el elemento 1, el elemento 2, o el elemento 3, por lo que todos se marcan como *UNKNOWN*. El algoritmo no permitió llegar a un acuerdo.

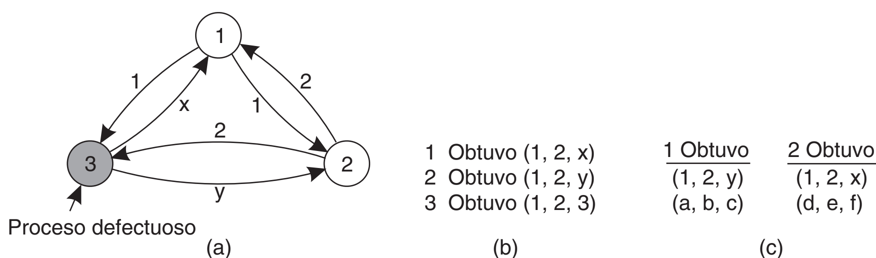


Figura 8-6. Lo mismo que en la figura 8-5, excepto que ahora hay dos procesos correctos y uno defectuoso.

En su artículo, Lamport y colaboradores (1982) demostraron que en un sistema con k procesos defectuosos, el acuerdo se puede lograr sólo si están presentes $2k + 1$ procesos que están funcionando correctamente, para un total de $3k + 1$. Dicho de otra manera, el acuerdo es posible sólo si *más* de dos tercios de los procesos están funcionando apropiadamente.

Otra forma de ver este problema es como sigue. Básicamente, lo que debemos lograr es una votación mayoritaria entre un grupo de procesos no defectuosos sin importar si también existen procesos defectuosos entre ellos. Si hay k procesos defectuosos, se tiene que garantizar que su voto, junto con el de cualesquiera procesos correctos que han sido engañados por los defectuosos, siga correspondiendo al voto mayoritario de los procesos no defectuosos. Con $2k + 1$ procesos no defectuosos, esto se puede lograr requiriendo que se llegue a un acuerdo sólo si más de dos tercios de los votos son los mismos. Es decir, si más de dos tercios de los procesos están de acuerdo en la misma decisión, esta decisión corresponde al mismo voto mayoritario realizado por el grupo de procesos no defectuosos.

Sin embargo, llegar a un acuerdo puede ser incluso peor. Fischer y colaboradores (1985) demostraron que en un sistema distribuido en el cual no se puede garantizar que los mensajes serán enviados dentro de un tiempo finito conocido, no es posible llegar a un acuerdo si incluso un proceso está defectuoso (aun cuando dicho proceso haya fallado silenciosamente). El problema con

sistemas como ese es que los procesos arbitrariamente lentos son indistinguibles de los procesos congelados (es decir, no se puede distinguir el inactivo del activo). Se conocen muchos otros resultados teóricos sobre cuándo es o no posible un acuerdo. En Barborak y colaboradores (1993) y Turek y Shasha (1992) se estudian estos resultados.

Es importante advertir que los esquemas descritos hasta ahora asumen que los nodos son o bizantinos o colaboradores. Lo segundo no siempre puede ser simplemente asumido cuando los procesos son de diferentes dominios administrativos. En ese caso, es muy probable que exhiban un comportamiento *racional*, por ejemplo, reportando tiempos fuera cuando hacerlo resulte más barato que ejecutar una operación de actualización. Como ocuparse de estos casos no es trivial, un primer paso hacia una solución se captura en la forma de **tolerancia a fallas BAR**, lo cual significa bizantino, altruismo, y racionalidad. La tolerancia a las fallas BAR se describe en Aiyer y colaboradores (2005).

8.2.4 Detección de fallas

Puede ser que lo expuesto hasta ahora deje claro que para enmascarar apropiadamente las fallas, en general también se requiere detectarlas. La detección de fallas es una de las piedras angulares de la tolerancia a fallas en sistemas distribuidos. A lo que se reduce esto es a que para un grupo de procesos, los miembros no defectuosos deberán ser capaces de decidir quién sigue siendo miembro y quién no. En otros términos, debemos ser capaces de detectar cuando un miembro ha fallado.

Cuando se trata de detectar fallas de procesos, en esencia existen sólo dos mecanismos. O los procesos se envían activamente mensajes del tipo "¿estás activo?" entre sí (para los cuales desde luego esperan una respuesta) o esperan pasivamente hasta que lleguen mensajes de los diferentes procesos. Lo segundo tiene sentido sólo cuando se puede garantizar que existe suficiente comunicación entre los procesos. En la práctica, por lo general se envía activamente un **ping** a los procesos.

Existe una enorme cantidad de trabajo teórico sobre detectores de fallas. A lo que se reduce es a la utilización de un mecanismo de tiempo fuera para verificar si un proceso ha fallado. En situaciones reales, existen dos problemas importantes con esta forma de proceder. En primer lugar, debido a las redes no confiables, afirmar simplemente que un proceso ha fallado porque no responde a un mensaje ping puede ser erróneo. En otros términos, es bastante fácil generar falsos positivos. Si un falso positivo tiene el efecto de que un proceso perfectamente saludable es eliminado de una lista de membresía, entonces claramente algo se está haciendo mal.

Otro problema serio es que los tiempos fuera ocurren con suma facilidad. Como lo señala Birman (2005), existen pocos trabajos sobre la construcción de subsistemas de detección de fallas apropiados que tomen en cuenta algo más que la simple falta de respuesta a un mensaje. Esta afirmación es aún más evidente cuando se consideran sistemas distribuidos utilizados en la industria.

Existen varios temas a tomar en cuenta cuando se diseña un subsistema de detección de fallas [vea también Zhuang y cols. (2005)]. Por ejemplo, la detección de fallas puede ocurrir mediante conversaciones en las que cada nodo informa con regularidad que sigue activo y funcionando. Como ya mencionamos, una alternativa es permitir que se sondeen activamente entre sí.

La detección de fallas también puede ocurrir como un efecto colateral del intercambio regular de información con los vecinos, igual que en el caso de la diseminación de información basada en gossip (la cual se analizó en el capítulo 4). Este enfoque también se adopta esencialmente en Obduro (Vogels, 2003): los procesos informan periódicamente acerca de su disponibilidad de servicio. Esta información se disemina gradualmente por la red a través de charlas. Con el tiempo, cada proceso se enterará de la existencia de todos los demás procesos pero, aún más importante, tendrá suficiente información localmente disponible para decidir si un proceso ha fallado o no. Un miembro para el cual la información sobre disponibilidad es obsoleta, presumiblemente ha fallado.

Otro tema importante es que un subsistema de detección de fallas idealmente deberá ser capaz de distinguir fallas de red de fallas de nodos. Una forma de abordar este problema es no permitir que un solo nodo decida si uno de sus vecinos falló. En vez de eso, cuando un nodo observa que no hay respuesta a un mensaje ping, solicita a otros vecinos ver si pueden comunicarse con el nodo que presumiblemente falló. Desde luego, la información positiva también puede compartirse: si un nodo sigue activo, esa información puede ser remitida a otras partes interesadas (las cuales pueden estar detectando una falla de enlace con el nodo sospechoso).

Esto conduce a otro tema clave: cuando se detecta la falla de un miembro, ¿cómo se deberá informarlo a otros procesos no defectuosos? Un método simple y un tanto radical es el que se sigue en FUSE (Dunagan y cols., 2004). En FUSE, los procesos pueden unirse en un grupo que comprenda una red de área amplia. Los miembros del grupo crean un árbol expansivo (*spanning tree*) utilizado para monitorear los miembros que fallan. Los miembros envían pings a sus vecinos. Cuando un vecino no responde, el nodo que envió el ping cambia de inmediato a un estado en el cual tampoco responderá a los pings enviados por otros nodos. Por recursión, se ve que la falla de un solo nodo es informada rápidamente al grupo. FUSE no experimenta muchas fallas de enlace por la simple razón de que utiliza conexiones TCP punto a punto entre los miembros del grupo.

8.3 COMUNICACIÓN CONFIABLE ENTRE CLIENTE Y SERVIDOR

En muchos casos, la tolerancia a fallas en sistemas distribuidos se concentra en procesos defectuosos. Sin embargo también se tienen que considerar las fallas de comunicación. Los modelos de fallas analizados previamente aquí también son válidos en su mayoría para canales de comunicación. En particular, un canal de comunicación puede presentar fallas por congelación, omisión, temporización, y arbitrarias. En la práctica, cuando se construyen canales de comunicación confiables, se presta especial atención al ocultamiento de fallas por congelación y omisión. Las fallas arbitrarias pueden presentarse como mensajes duplicados, a consecuencia de que en una red de computadoras los mensajes pueden almacenarse en bufer durante mucho tiempo, y son reenviados a la red después de que el remitente original emite una orden de retransmisión [vea, por ejemplo, Tanenbaum (2003)].

8.3.1 Comunicación punto a punto

En muchos sistemas distribuidos, la comunicación punto a punto confiable se establece por medio de un protocolo de transporte confiable, tal como el TCP. Éste oculta las fallas por omisión, las cuales se presentan en la forma de mensajes perdidos, por medio de reconocimientos y retransmisiones. Tales fallas permanecen completamente ocultas de un cliente TCP.

No obstante, las fallas por congelación no se ocultan. Una falla por congelación puede ocurrir (por cualquier razón) cuando una conexión TCP se interrumpe repentinamente de modo que no pueden transmitirse más mensajes a través del canal. En la mayoría de los casos, el cliente es informado de que el canal se congeló porque presentó una excepción. La única forma de ocultar ese tipo de fallas es permitir al sistema distribuido establecer de forma automática una nueva conexión, reenviando simplemente una petición de conexión. La suposición subyacente es que el otro lado puede, o se encuentra otra vez en la situación de poder, responder a tales peticiones.

8.3.2 Semántica RPC en presencia de fallas

A continuación se examinará más a fondo la comunicación entre cliente y servidor cuando se utilizan medios de comunicación de alto nivel tales como llamadas a procedimientos remotos (RPC, del inglés *Remote Procedure Calls*). El objetivo de las RPC es ocultar la comunicación de tal forma que las llamadas a procedimientos remotos parezcan locales. Con pocas excepciones, hasta ahora se ha estado cerca de resolver el problema. Por cierto, en tanto el cliente y el servidor funcionen bien, las RPC hacen bien su trabajo. El problema ocurre cuando surgen errores. Es entonces que las diferencias entre las llamadas locales y las remotas no siempre son fáciles de ocultar.

Para estructurar nuestro análisis, haremos una distinción entre las cinco clases de fallas diferentes que pueden ocurrir en sistemas RPC, como sigue:

1. El cliente es incapaz de localizar el servidor.
2. El mensaje de solicitud (petición) del cliente al servidor se pierde.
3. El servidor se congela después de recibir una petición.
4. El mensaje de respuesta del servidor al cliente se pierde.
5. El cliente se congela después de enviar una solicitud.

Cada una de estas categorías plantea problemas diferentes y requiere soluciones diferentes.

El cliente no puede localizar el servidor

Por principio de cuentas, suele suceder que el cliente no puede localizar un servidor adecuado. Todos los servidores podrían estar apagados, por ejemplo. Alternativamente, supongamos que el cliente fue compilado con una versión particular del resguardo de cliente y que el binario no se ha utilizado durante un lapso de tiempo considerable. Entretanto, el servidor evoluciona y se instala

una nueva versión de la interfaz; se generan y ponen en uso nuevos resguardos. Cuando finalmente se ejecuta el cliente, el conector de archivos será incapaz de mantenerse a la par del servidor y reportará una falla. Si bien este mecanismo es usado para proteger al cliente contra un intento accidental de comunicarse con un servidor que pudiera no estar de acuerdo con él en función de qué parámetros se requieran o con lo que se supone que hace, el problema continúa siendo cómo se deberá manejar la falla.

Una posible solución es lograr que el error haga surgir una **excepción**. En algunos lenguajes (por ejemplo en Java), los programadores pueden escribir procedimientos especiales que sean invocados por errores específicos, tal como la división entre cero. En C, es posible utilizar manejadores de señales para este propósito. Es decir, se podría definir un nuevo tipo de señal *SIGNO-SERVER*, la cual se manejaría de igual modo que las demás señales.

Este método, también, tiene ventajas. Por principio de cuentas, no todos los lenguajes tienen excepciones o señales. Otro punto es que al tratar de escribir una excepción o manejador de señal se destruye la transparencia que se ha estado tratando de lograr. Supongamos que usted es un programador y que su jefa le pide escribir el procedimiento `sum`. Usted sonríe y le dice que lo escribirá, probará, y documentará en cinco minutos. Entonces ella menciona que usted también deberá escribir un manejador de excepciones, por si el procedimiento no está listo el mismo día. En este punto es muy difícil mantener la ilusión de que los procedimientos remotos no son diferentes de los locales, puesto que escribir un manejador de excepción para "El servidor no puede ser localizado" sería una solicitud un tanto inusual en un sistema de un solo procesador. Todo sea por la transparencia.

Pérdida de mensajes de solicitud

El segundo elemento en la lista tiene que ver con los mensajes de solicitud perdidos. Es el más fácil de abordar: sólo debe lograrse que el sistema operativo o resguardo de cliente inicie un cronómetro cuando envíe la solicitud. Si el cronómetro expira antes de que regrese una respuesta o reconocimiento, el mensaje es reenviado. Si el mensaje realmente se perdió, el servidor no será capaz de distinguir entre la retransmisión y el envío original, y todo funcionará bien. A menos que, desde luego, muchos mensajes se pierdan de tal suerte que el cliente desista y falsamente concluya que el servidor está fuera de servicio, en cuyo caso se vuelve a la situación en que "El servidor no puede ser localizado". Si la solicitud no se ha perdido, lo único que debe hacerse es permitir que el servidor detecte que se trata de una retransmisión. Desafortunadamente, esto no resulta muy simple, como explicaremos al estudiar las respuestas perdidas.

Congelación del servidor

La siguiente falla en la lista, es la congelación de un servidor. La secuencia normal de eventos en un servidor se muestra en la figura 8-7(a). Llega una solicitud, es atendida, y se envía una respuesta. Ahora consideremos la figura 8-7(b). Llega una solicitud y es atendida, como antes, pero el servidor se congela antes de poder responder. Por último, veamos la figura 8-7(c). De nueva cuenta llega una solicitud, pero entonces el servidor se congela incluso antes de poder atenderla. Y, desde luego, no envía ninguna respuesta.

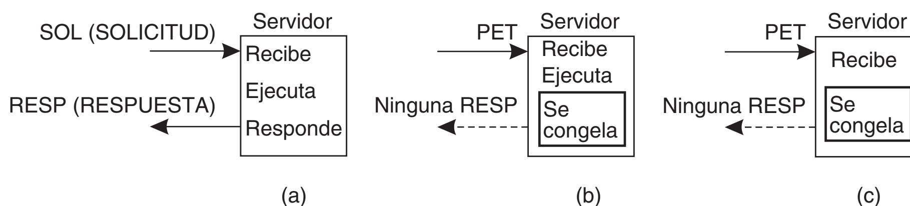


Figura 8-7. Un servidor en el caso de comunicación cliente-servidor. (a) El caso normal. (b) Congelación después de la ejecución. (c) Congelación antes de la ejecución.

La parte incómoda de la figura 8-7 es que el tratamiento correcto es diferente en (b) y en (c). En (b), el sistema tiene que volver a reportar la falla al cliente (por ejemplo, emitiendo una excepción), en tanto que en (c) sólo se puede retransmitir la solicitud. El problema es que el sistema operativo del cliente no puede distinguir cuál es cuál. Todo lo que sabe es que su cronómetro ha expirado.

Existen tres escuelas de pensamiento sobre lo que debe hacerse en este caso (Spector, 1982). Una filosofía es esperar hasta que el servidor se reinicie (o se reconecte a un nuevo servidor) y volver a intentar la operación. La idea es seguir intentándolo hasta que se haya recibido una respuesta para luego entregarla al cliente. Esta técnica se llama **semántica de por lo menos una vez** y garantiza que la RPC se haya realizado al menos una vez, aunque posiblemente más veces.

La segunda filosofía cede de inmediato y reporta la falla. Esta forma se llama **semántica a lo más una vez** y garantiza que la RPC se ha realizado cuando mucho una vez, pero posiblemente ninguna en absoluto.

La tercera filosofía es no garantizar nada. Cuando un servidor se congela, el cliente no obtiene ninguna ayuda y nada de promesas sobre lo que sucedió. La RPC puede haber sido realizada en cualquier parte desde cero hasta un gran número de veces. La principal virtud de este esquema es que resulta fácil de implementar.

Ninguna de estas filosofías es totalmente atractiva. Lo que se requeriría sería una **semántica de exactamente una vez**, aunque en general no hay ninguna forma de solucionar esta cuestión. Imagínese que la operación remota consiste en imprimir algún texto y que el servidor remoto envía un mensaje de terminación al cliente cuanto el texto ya fue impreso. Suponga también que cuando un cliente emite una petición, recibe un acuse de recibo de que la solicitud fue entregada al servidor. El servidor puede seguir entonces dos estrategias: enviar un mensaje de terminación exactamente antes de que le ordene a la impresora realizar su trabajo, o enviarlo después de que el texto haya sido impreso.

Supongamos que el servidor se congela y luego se recupera. Avisa a todos los clientes que se acaba de congelar pero que ya está en marcha y funcionando otra vez. El problema es que el cliente no sabe si su solicitud imprimir un texto en efecto fue realizada.

Existen cuatro estrategias que el cliente puede seguir. Primera, el cliente puede decidir *nunca* reemitir una solicitud, con el riesgo de que el texto no sea impreso. Segunda, puede decidir reemitir *siempre* una solicitud, pero esto lleva a que su texto se imprima dos veces. Tercera, puede decidir

reemitir una solicitud sólo si aún no ha recibido un acuse de recibo de que su solicitud de impresión había sido entregada al servidor. En ese caso, el cliente confía en que el servidor se congeló antes de que la solicitud pudiera ser entregada. La cuarta y última estrategia es reemitir una solicitud siempre y cuando se haya recibido un acuse de recibo de la solicitud de impresión.

Con dos estrategias para el servidor y cuatro para el cliente, existe un total de seis combinaciones a considerar. Desafortunadamente, ninguna es satisfactoria. Para explicarlo, observemos que pueden suceder tres eventos en el servidor: enviar el mensaje de terminación (M), imprimir el texto (P), y congelación (C). Estos eventos pueden ocurrir en seis órdenes diferentes.

1. $M \rightarrow P \rightarrow C$: Ocurre una congelación después de enviar el mensaje de terminación y de imprimir el texto.
2. $M \rightarrow C$ ($\rightarrow P$): Ocurre una congelación después de enviar el mensaje de terminación, pero antes de que el texto pueda ser impreso.
3. $P \rightarrow M \rightarrow C$: Ocurre una congelación después de enviar el mensaje de terminación y de imprimir el texto.
4. $P \rightarrow C$ ($\rightarrow M$): El texto se imprime, tras de lo cual se congela el servidor antes de que el mensaje de terminación pueda ser enviado.
5. C ($\rightarrow P \rightarrow M$): Ocurre una congelación antes de que el servidor pueda hacer algo.
6. C ($\rightarrow M \rightarrow P$): Ocurre una congelación antes de que el servidor pueda hacer algo.

Los paréntesis indican un evento que ya no puede suceder porque el servidor ya se congeló. La figura 8-8 muestra todas las posibles combinaciones. Se advierte de inmediato que no existe ninguna combinación de estrategia de cliente y estrategia de servidor que funcione correctamente en todas las secuencias de eventos posibles. El renglón inferior indica que el cliente nunca puede saber si el servidor se congeló un poco antes o después de haber impreso el texto.

Cliente	**Servidor**					
	Estrategia M → P			**Estrategia P → M**		
Estrategia de reemisión	**MPC**	**MC (P)**	**C(MP)**	**PMC**	**PC(M)**	**C(PM)**
Siempre	DUP	OK	OK	DUP	DUP	OK
Nunca	OK	CERO	CERO	OK	OK	CERO
Sólo cuando sea confirmado	DUP	OK	CERO	DUP	OK	CERO
Sólo cuando no es confirmado	OK	CERO	OK	OK	DUP	OK

OK = El texto se imprime una vez
DUP = El texto se imprime dos veces
CERO = El texto no se imprime para nada

Figura 8-8. Diferentes combinaciones de estrategias de cliente y servidor en presencia de congelaciones del servidor.

En suma, la posibilidad de que un servidor se congele cambia radicalmente la naturaleza de la RPC y distingue con claridad los sistemas de un solo procesador de los sistemas distribuidos. En el primer caso, una congelación de servidor también implica una congelación del cliente, por lo que la recuperación no es posible ni necesaria. En el segundo caso, es tanto posible como necesario actuar.

Pérdida de mensajes de respuesta

Las respuestas perdidas también son difíciles de manejar. La solución más evidente es confiar de nuevo en un cronómetro que haya sido ajustado por el sistema operativo del cliente. Si no llega ninguna respuesta dentro de un lapso de tiempo razonable, simplemente se envía la solicitud una vez más. El problema con esta solución es que el cliente en realidad no está seguro de por qué no hubo respuesta. ¿Respondió el servidor y se perdió la respuesta, o simplemente el servidor está lento? Ello puede hacer la diferencia.

En particular, algunas operaciones pueden ser repetidas con seguridad tan frecuentemente como sea necesario sin que ocurra ningún daño. Una petición tal como preguntar por los primeros 1 024 bytes de un archivo no tiene efectos secundarios y puede ser ejecutada tan a menudo como se necesite sin ningún perjuicio. Se dice que una solicitud que tiene esta propiedad es **idempotente**.

Consideremos ahora una solicitud al servidor de un banco de transferir un millón de dólares de una cuenta a otra. Si la solicitud llega y es realizada, pero se pierde la respuesta, el cliente no lo sabrá y volverá a transmitir el mensaje. El servidor del banco interpretará esta solicitud como nueva y también la realizará. Dos millones de dólares serán transferidos. Esperemos que la respuesta no se pierda 10 veces. La transferencia de dinero no es idempotente.

Una forma de resolver este problema es tratar de estructurar todas las solicitudes en una forma idempotente. En la práctica, sin embargo, muchas solicitudes (por ejemplo, transferir dinero) son inherentemente no idempotentes, por lo que se requiere de algo más. Otro método es hacer que el cliente asigne un número de secuencia a cada solicitud. Haciendo que el servidor lleve la cuenta del número de secuencia más recientemente recibido de cada cliente que lo está utilizando, el servidor puede distinguir entre una solicitud original y una retransmisión y puede rehusarse a realizar cualquier solicitud por segunda vez. No obstante, el servidor aún tendrá que seguir respondiendo al cliente. Observemos que este enfoque requiere que el servidor mantenga administración sobre cada cliente. Además, no queda claro por cuánto tiempo hay que mantener esta administración. Una protección adicional es utilizar un bit en el encabezado del mensaje para diferenciar las solicitudes iniciales de las retransmisiones (siendo la idea que esto siempre asegure la ejecución de una petición original; las retransmisiones pueden requerir más cuidado).

Congelaciones de cliente

El elemento final en la lista de fallas es la congelación del cliente. ¿Qué sucede si un cliente envía una solicitud a un servidor de que realice un trabajo y se congela antes de que le conteste? En ese momento se activa un cálculo y no hay ningún padre en espera de la respuesta. Semejante cálculo indeseado se llama **huérfano**.

Los huérfanos suelen provocar varios problemas que interfieren con la operación normal de un sistema. Por lo menos, desperdician ciclos de CPU. También pueden bloquear archivos o bien atarlos a recursos valiosos. Por último, si el cliente se reinicia y realiza la RPC, pero la respuesta del huérfano llega de inmediato, puede haber confusión.

¿Qué se puede hacer con los huérfanos? Nelson (1981) propuso cuatro soluciones. En la solución 1, antes de que el resguardo de un cliente envíe un mensaje de RPC, elabora una entrada de bitácora para informar lo que está a punto de hacer. La bitácora se mantiene en el disco o en algún otro medio que sobreviva a las congelaciones. Después del reinicio, se revisa la bitácora y el huérfano es explícitamente eliminado. Esta solución se conoce como **exterminio de huérfanos**.

La desventaja de este esquema es el horrendo gasto de escribir una bitácora de disco por cada RPC. Por si fuera poco, incluso es posible que no funcione puesto que los propios huérfanos pueden elaborar RPC, con lo que se crean **huérfanos nietos** o más descendientes difíciles o imposibles de localizar. Por último, puede que la red resulte particionada debido a una puerta de enlace (Gateway) defectuosa, de modo que resultará imposible eliminarlos aunque puedan ser localizados. En definitiva, éste no es un enfoque promisorio.

En la solución 2, llamada **reencarnación**, todos estos problemas se resuelven sin tener que escribir registros de disco. La forma en que funciona es dividiendo el tiempo en épocas numeradas en secuencia. Cuando un cliente se reinicia, transmite un mensaje a todas las máquinas para indicarles el inicio de una nueva época. Cuando tal transmisión llega, todos los cálculos realizados a nombre de dicho cliente son eliminados. Desde luego, si la red está particionada, puede que algunos huérfanos sobrevivan. Desafortunadamente, sin embargo, cuando se vuelven a reportar, sus respuestas contendrán un número de época obsoleto, ello facilita su detección.

La solución 3 es una variante de esta idea, pero un poco menos draconiana. Se llama **reencarnación benévola**. Cuando llega una transmisión de época, cada máquina ve si tiene cálculos remotos ejecutándose localmente, y de ser así, hace lo que puede para localizar a sus propietarios. Sólo si los propietarios no pueden ser localizados en ninguna parte el cálculo se elimina.

Por último, tenemos la solución 4, **expiración**, en la cual a cada RPC se le asigna un cantidad estándar de tiempo, T, para que realice el trabajo. Si no puede terminarlo, explícitamente debe solicitar otra cantidad de tiempo, lo cual es una molestia. Por otra parte, si después de congelarse el cliente espera cierto tiempo T antes de volverse a iniciar, con seguridad habrán desaparecido todos los huérfanos. El problema a resolver en este caso es seleccionar un valor razonable de T de cara a las RPC con requerimientos desatinadamente diferentes.

En la práctica, todos estos métodos son rudimentarios e indeseables. Peor aún, la eliminación de un huérfano puede acarrear consecuencias impredecibles. Por ejemplo, supongamos que un huérfano ha obtenido derechos sobre uno o más archivos o registros de base de datos. Si el huérfano es eliminado de repente, estos derechos pueden permanecer por siempre. También, un huérfano pudiera haber hecho ya entradas en varias colas remotas para iniciar otros procesos en cierto tiempo futuro, así que incluso la eliminación del huérfano puede no eliminar todos sus rastros. Es más, incluso puede haberse reiniciado, con consecuencias impredecibles. Panzieri y Shrivastava (1988) se ocupan con más detalle de la eliminación de huérfanos.

8.4 COMUNICACIÓN DE GRUPO CONFIABLE

Al considerar lo importante que es la atenuación de un proceso por replicación, no es de sorprender que los servicios de multitransmisión confiables también sean importantes. Por desgracia, la multitransmisión confiable resulta ser sorprendentemente complicada. En esta sección, examinamos más a fondo los temas implicados en la entrega confiable de mensajes a un grupo de procesos.

8.4.1 Esquemas de multitransmisión básicos confiables

Aunque la mayoría de las capas de transporte ofrecen confiables canales de comunicación punto a punto, rara vez ofrecen una comunicación confiable a un conjunto de procesos. Su mejor oferta es permitir que cada proceso establezca una conexión punto a punto con cualquier otro proceso con el que desee comunicarse. Desde luego, semejante organización no es muy eficiente ya que puede desperdiciar ancho de banda de la red. Sin embargo, cuando el número de procesos es pequeño, lograr la confiabilidad mediante múltiples canales punto a punto confiables es una solución simple y directa.

Para ir más allá de este caso simple, tenemos que definir con precisión qué es la multitransmisión confiable. Intuitivamente, significa que un mensaje enviado a un grupo de procesos deberá ser entregado a cada uno de los miembros de dicho grupo. Sin embargo, ¿qué pasa si durante la comunicación un proceso se une al grupo? ¿Deberá recibir también el mensaje? Asimismo, deberemos determinar qué sucede si un proceso (remitente) se congela durante la comunicación.

Para cubrir tales situaciones, debemos distinguir entre comunicación confiable en presencia de procesos defectuosos y comunicación confiable cuando se supone que los procesos están operando correctamente. En el primer caso, se considera que la multitransmisión es confiable cuando puede garantizar que todos los miembros no defectuosos del grupo recibirán el mensaje. La parte engañosa es que deberá llegarse a un acuerdo sobre cómo se ve en realidad el grupo antes de que un mensaje pueda ser entregado, además de implementar varias restricciones en cuanto al orden de entrega. Cuando más adelante volvemos a tocar estos temas al abordar las multitransmisiones atómicas.

La situación se simplifica cuando suponemos que existe un acuerdo sobre quién es miembro del grupo y quién no lo es. En particular, al suponer que los procesos no fallan, y que no se unen al grupo o lo dejan mientras la comunicación se está llevando a cabo, multitransmisión confiable significa simplemente que todo mensaje deberá ser entregado a cada miembro actual del grupo. En el caso más simple, no existe ningún requerimiento de que todos los miembros del grupo reciban mensajes en el mismo orden, aunque en ocasiones se requiere cumplir con esta característica.

Esta forma más débil de multitransmisión confiable es relativamente fácil de implementar, sujeta de nuevo a la condición de que el número de destinatarios esté limitado. Consideremos el caso en que un solo remitente desea enviar un mensaje a varios destinatarios. Supongamos que el sistema de comunicación empleando multitransmisión ofrece sólo multitransmisión confiable, lo cual

significa que un mensaje multitransmitido puede perderse en parte y ser entregado a algunos, pero no a todos los remitentes pensados.

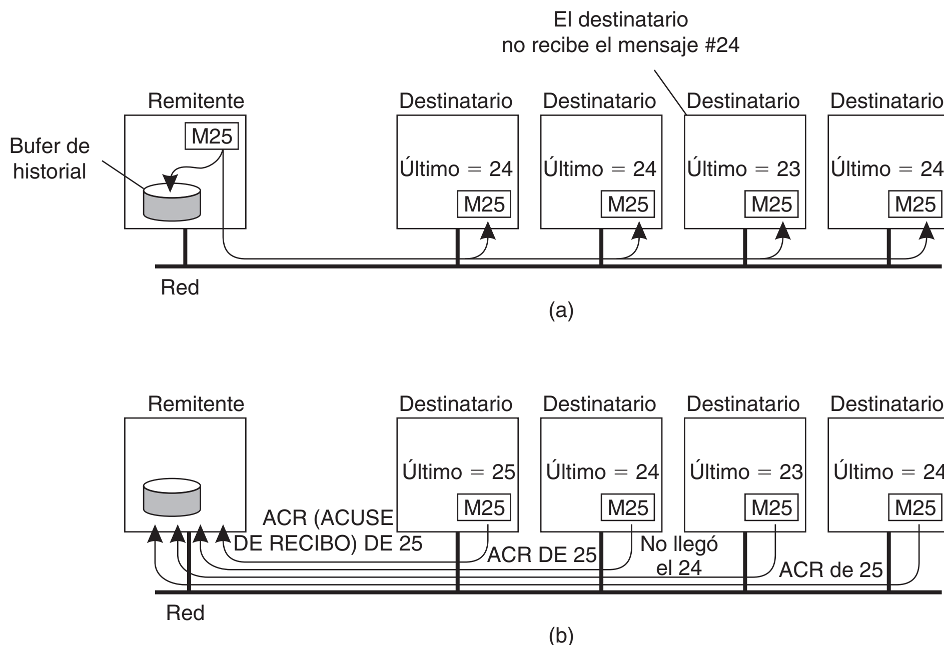


Figura 8-9. Una solución simple para la multitransmisión confiable cuando todos los destinatarios son conocidos y se asume que no fallan. (a) Transmisión de mensajes. (b) Reporte de retroalimentación.

En la figura 8-9 se muestra una solución simple. El proceso remitente asigna un número de secuencia a cada mensaje que se multitransmite. Se supone que los mensajes se reciben en el orden en que fueron enviados. De este modo, es fácil que un destinatario detecte que no recibió un mensaje. Cada mensaje multitransmitido se guarda localmente en un bufer de historial en el remitente. Suponiendo que el remitente conoce a los destinatarios, éste simplemente conserva el mensaje en su búfer de historial hasta que cada destinatario haya devuelto un acuse de recibo. Si un destinatario detecta que se perdió un mensaje, regresa un acuse de recibo negativo, solicitando al remitente que retransmita el mensaje. De otro modo, el remitente puede retransmitir automáticamente el mensaje cuando no haya recibido todos los acuses de recibo dentro de cierto tiempo.

Existen varios cambios de diseño que pueden hacerse. Por ejemplo, para reducir el número de mensajes devueltos al remitente, los acuses de recibo podrían ser incorporados a otros mensajes. Además, la retransmisión de un mensaje puede hacerse por medio de comunicación punto a punto a cada proceso solicitante, o utilizando un mensaje monotransmitido a todos los procesos. En Defago y colaboradores (2004) se encuentra un estudio extenso y detallado de multitransmisiones de orden total.

8.4.2 Escalabilidad en multitransmisión confiable

El problema principal con el esquema de multitransmisión confiable que se acaba de describir es que no puede soportar un gran número de destinatarios. Si existen *N* destinatarios, el remitente debe estar preparado para aceptar por lo menos *N* acuses de recibo. Con muchos destinatarios, el remitente puede verse abrumado por los mensajes de retroalimentación, ello también se conoce como implosión de retroalimentación. Además, puede que también se requiera tomar en cuenta que los destinatarios se encuentran esparcidos por toda la red de área amplia.

Una solución a este problema es no hacer que los destinatarios confirmen la recepción de un mensaje. En cambio, un destinatario devuelve un mensaje de retroalimentación sólo para informar que el remitente no envió un mensaje. Regresando sólo acuses de recibo negativos generalmente se puede a trabajar a mayor escala [vea, por ejemplo, Towsley y cols. (1997)], aunque no se pueden dar garantías estrictas de que nunca ocurrirán implosiones de retroalimentación.

Otro problema con la devolución de sólo acuses de recibo negativos es que el remitente, en teoría, se verá obligado a conservar por siempre un mensaje en un bufer de historial. Como puede ser que el remitente nunca sepa si un mensaje ha sido entregado correctamente a todos los destinatarios, siempre deberá estar preparado por si un destinatario le solicita la retransmisión de un mensaje viejo. En la práctica, el remitente borrará un mensaje de su bufer de historial después de transcurrido cierto tiempo, para evitar que el bufer se sobrecargue. No obstante, la eliminación de un mensaje se realiza con el riesgo de que una solicitud de retransmisión no pueda ser atendida.

Existen varias propuestas para implementar multitransmisión confiable escalable. En Levine y García-Luna-Aceves (1998) se comparan diferentes esquemas. A continuación presentamos brevemente dos enfoques muy distintos que son representativos de muchas soluciones existentes.

Control de retroalimentación no jerárquico

El tema fundamental en relación con soluciones escalables para multitransmisión confiable es reducir el número de mensajes de retroalimentación devueltos al remitente. Un modelo popular usado en varias aplicaciones de área amplia es la **supresión de retroalimentación**. Este esquema sirve de fundamento al protocolo de **multitransmisión confiable escalable** (**SRM**, del inglés *Scalable Reliable Multicasting*) desarrollado por Floyd y colaboradores (1977), y funciona como sigue.

En primer lugar, en SRM los destinatarios nunca confirman la entrega exitosa de un mensaje multitransmitido, sino que informan sólo cuando no reciben un mensaje. El cómo detectar la pérdida de un mensaje se deja a la aplicación. Sólo se devuelven acuses de recibo negativos como retroalimentación. Siempre que un destinatario advierta que le falta un mensaje, *multitransmite* su retroalimentación al resto del grupo.

La multitransmisión mediante retroalimentación permite que otro miembro del grupo cancele su propia retroalimentación. Supongamos que varios destinatarios no reciben el mensaje *m*. Cada quien tendrá que devolver un acuse negativo al remitente, *S*, de modo que *m* pueda ser retransmitido. No obstante, si se supone que las retransmisiones siempre se multitransmiten a todo el grupo, es suficiente con que *S* reciba sólo una solicitud de retransmisión.

Por esta razón, un remitente *R* que no recibió un mensaje *m* programa un mensaje de retroalimentación con cierta demora aleatoria. Es decir, no envía la solicitud de retransmisión sino hasta que haya transcurrido cierto tiempo aleatorio. Si, en el ínterin, *R* recibe otra solicitud de retransmisión de *m*, *R* cancelará su propia retroalimentación, a sabiendas de que *m* será retransmitido en breve. De esta manera, idealmente, *S* recibirá sólo un mensaje de retroalimentación, el que a su vez retransmite luego *m*. Este esquema se muestra en la figura 8-10.

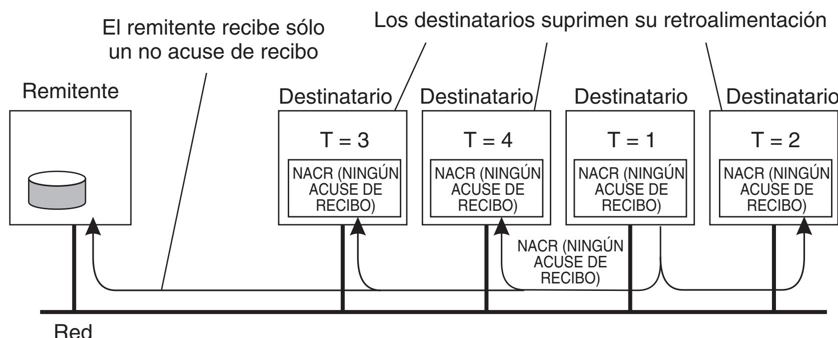


Figura 8-10. Varios destinatarios han programado una solicitud de retransmisión, pero la primera solicitud conduce a la supresión de las demás.

La supresión de retroalimentación ha mostrado escalar razonablemente bien, y ha sido utilizada como el mecanismo fundamental en varias aplicaciones colaborativas en internet tal como un pizarrón compartido (*Shared whiteboard*) compartido. No obstante, este método también presenta varios problemas serios. En primer lugar, garantizar que el remitente reciba sólo una solicitud de retransmisión requiere una programación razonablemente precisa de los mensajes de retroalimentación por parte de cada destinatario. De lo contrario, muchos destinatarios seguirán enviando su retroalimentación al mismo tiempo. No es tan fácil ajustar los cronómetros como corresponde en un grupo de procesos dispersos a través de una red de área amplia.

Otro problema es que la multitransmisión de la retroalimentación también interrumpe aquellos procesos que recibieron el mensaje exitosamente. En otros términos, otros destinatarios se ven obligados a recibir y procesar mensajes inútiles para ellos. La única solución a este problema es permitir que los destinatarios que no hayan recibido el mensaje *m* se unan a un grupo de multitransmisión distinto para *m*, como se explica en Kasera y colaboradores (1997). Desafortunadamente, esta solución requiere que los grupos se manejen de una forma altamente eficiente, lo cual es difícil de lograr en un sistema de área amplia. Un mejor enfoque es, por consiguiente, permitir que los destinatarios propensos a no recibir el mensaje *m* hagan equipo y compartan el mismo canal de multitransmisión para retroalimentar y retransmitir mensajes. Detalles de este método se encuentran en Liu y colaboradores (1998).

Para incrementar la escalabilidad de SRM, es útil que los destinatarios ayuden en la recuperación local. En particular, si un destinatario que haya recibido con éxito el mensaje *m* recibe una solicitud para que lo retransmita, puede decidir multitransmitir *m* incluso antes de que el remitente

original reciba la solicitud de retransmisión. En Floyd y colaboradores (1997) y Liu y colaboradores (1998) se proporcionan más detalles.

Control de retroalimentación jerárquico

La supresión de retroalimentación como se acaba de describir es básicamente una solución no jerárquica. Sin embargo, lograr la escalabilidad para grupos de destinatarios muy grandes requiere la adopción de métodos jerárquicos. En esencia, una solución jerárquica para multitransmisión confiable funciona como se muestra en la figura 8-11.

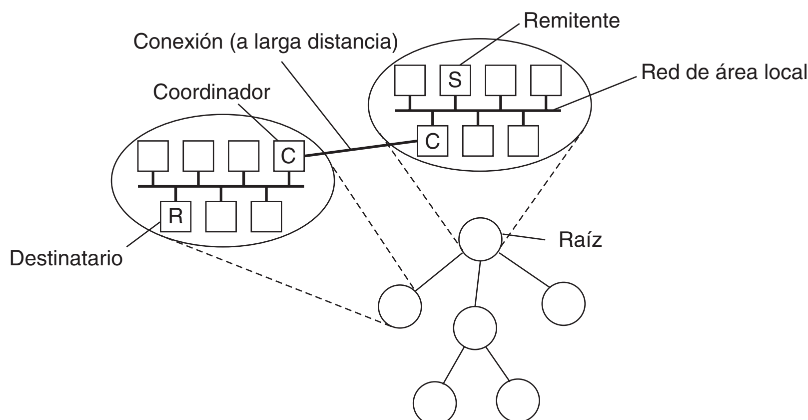


Figura 8-11. Esencia de la multitransmisión confiable jerárquica. Cada coordinador local remite el mensaje a sus hijos, y posteriormente se ocupa de las solicitudes de retransmisión.

Para simplificar las cosas, supongamos que existe sólo un remitente que debe transmitir mensajes a un grupo muy grande de destinatarios. El grupo de destinatarios se divide en varios subgrupos, ello posteriormente se organizan en un árbol. El subgrupo que contiene el remitente forma la raíz de su árbol. Dentro de cada subgrupo, se puede utilizar cualquier esquema de multitransmisión confiable que funcione para grupos pequeños.

Cada subgrupo designa un coordinador local, el cual es responsable de manejar las peticiones de retransmisión de los destinatarios contenidos en su subgrupo. El coordinador local dispondrá, por lo tanto, de su propio búfer de historial. Si el coordinador no envía un mensaje *m*, le pide al coordinador del subgrupo padre que lo retransmita. En un esquema de basado en acuses de recibo, un coordinador local envía un acuse de recibo a su padre si recibió el mensaje. Si un coordinador recibe acuses de recibo del mensaje *m* de todos los miembros que integran su subgrupo, y también de su hijo, puede eliminar *m* de su bufer de historial.

El problema principal con las soluciones jerárquicas es la construcción del árbol. En muchos casos, tiene que construirse dinámicamente. Una forma es utilizar el árbol de multitransmisión de la red subyacente, si existe alguno. En principio, el método consiste en mejorar entonces cada enrutador de multitransmisión en la capa de red de tal modo que actúe como coordinador local en la

forma que se acaba de describir. Desafortunadamente, en la práctica, tales adaptaciones en redes de computadoras existentes no son fáciles de realizar. Por estas razones, las soluciones de multitransmisión a nivel de aplicación abordadas en el capítulo 4 cada vez ganan más aceptación.

En conclusión, construir esquemas de multitransmisión que puedan escalarse a un número más grande de destinatarios a través de una red de área amplia es un problema difícil. No existe una solución única, y cada solución presenta nuevos problemas.

8.4.3 Multitransmisión atómica

A continuación regresamos a la situación donde se tiene que lograr una multitransmisión confiable en la presencia de fallas de proceso. En particular, lo que a menudo se requiere en un sistema distribuido es la garantía de que un mensaje sea entregado o a todos los procesos o a ninguno en absoluto. Además, en general, también se necesita que todos los mensajes sean entregados en el mismo orden a todos los procesos. Esto es conocido también como **problema de multitransmisión atómica**.

Para ver por qué es tan importante la atomicidad, consideremos una base de datos replicada construida como una aplicación encima de un sistema distribuido. El sistema distribuido ofrece servicios de multitransmisión confiables. En particular, permite construir grupos de procesos a los que los mensajes puedan ser enviados con toda confianza. La base de datos replicada se construye por consiguiente como un grupo de procesos, un proceso por cada réplica. Las operaciones actualizadas siempre son multitransmitidas a todas las réplicas y posteriormente se realizan a nivel local. En otros términos, suponemos que se utiliza un protocolo de replicación activa.

Supongamos que ahora se debe realizar una serie de actualizaciones, pero que durante la ejecución de una de ellas se congela una réplica. Por consiguiente, la actualización de dicha réplica se pierde aunque, por otra parte, se realiza correctamente en las demás réplicas.

Cuando una réplica recién congelada se recupera, en el mejor de los casos podrá recuperarse al mismo estado que tenía antes de la congelación; sin embargo, puede que se hayan perdido varias actualizaciones. En ese punto, resulta esencial que se ponga al día con las demás réplicas. Llevar la réplica al mismo estado de las demás requiere saber con exactitud qué operaciones se perdió y en qué orden se deben realizar éstas.

Supongamos ahora que el sistema distribuido subyacente soportaba multitransmisión atómica. En ese caso, la operación de actualización enviada a todas las réplicas justo antes de que una de ellas se congelara es realizada o en todas las réplicas no defectuosas o en ninguna. En particular, con multitransmisión atómica, la operación puede ser realizada por todas las réplicas que estén operando correctamente sólo si han llegado a un acuerdo en cuanto a la membresía del grupo. Es decir, la actualización se realiza si las réplicas restantes deciden que la réplica congelada ya no pertenece al grupo.

Cuando la réplica congelada se recupera, es obligada a unirse al grupo una vez más. No se le remitirán actualizaciones hasta que esté registrada otra vez como miembro. La unión al grupo requiere que su estado se encuentre a la par del resto de los miembros. En consecuencia, la multitransmisión atómica garantiza que los procesos no defectuosos mantendrán una vista consistente de la base de datos y obligarán a la reconciliación cuando una réplica se recupere y vuelva a unirse al grupo.

Sincronía virtual

La multitransmisión confiable en la presencia de fallas de proceso puede ser definida con precisión en función de grupos de procesos y cambios de membresía al grupo. Como antes, hacemos una distinción entre *recibir* y *entregar* un mensaje. En particular, adoptamos de nuevo un modelo en el cual el sistema distribuido se compone de una capa de comunicación, como se muestra en la figura 8-12. Dentro de esta capa de comunicación, se envían y reciben mensajes. Un mensaje recibido se guarda localmente en el búfer en la capa de comunicación hasta que pueda ser entregado a la aplicación colocada lógicamente en una capa más alta.

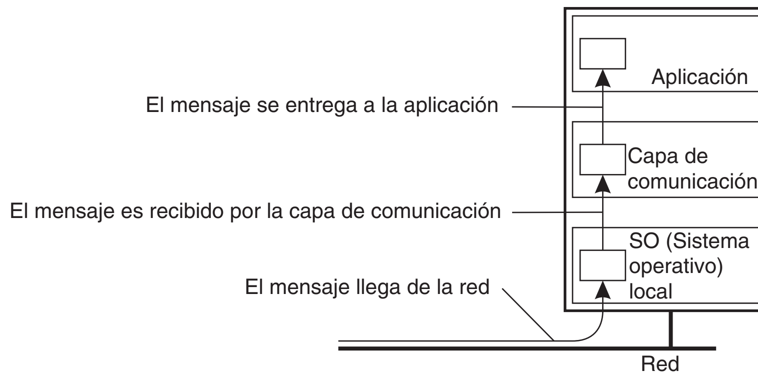


Figura 8-12. Organización lógica de un sistema distribuido para distinguir entre recepción y entrega de un mensaje.

La idea completa de la multitransmisión atómica es que un mensaje multitransmitido *m* está asociado únicamente con una lista de procesos a los que deberá ser entregado. Esta lista de entrega corresponde a una **vista de grupo**, es decir, la vista del conjunto de procesos contenidos en el grupo, la que tenía el remitente en el momento que el mensaje *m* fue multitransmitido. Una observación importante es que cada proceso que aparece en la lista tiene la misma vista. Es decir, todos los procesos deberán estar de acuerdo en que *m* deberá ser entregado a cada uno de ellos y no a otros procesos.

Supongamos ahora que el mensaje *m* es multitransmitido en el momento en que el remitente tiene una vista del grupo *G*. Además, asuma que mientras la multitransmisión está ocurriendo, otro proceso se une al grupo o lo abandona. Este cambio de membresía al grupo naturalmente que es anunciado a todos los procesos incluidos en *G*. Expresado en una forma un tanto diferente, ocurre un **cambio de vista** cuando se transmite, a múltiples destinatarios, un mensaje *vc* para informar la unión a, o el abandono de, un proceso. Ahora se tienen dos mensajes multitransmitidos en tránsito al mismo tiempo: *m* y *vc*. Lo que se debe garantizar es que *m* sea entregado a todos los procesos incluidos en *G* antes de que cada uno de ellos reciba el mensaje *vc* o que *m* no sea entregado en absoluto. Observemos que este requerimiento es comparable en parte a la multitransmisión totalmente ordenada, la cual estudiamos en el capítulo 6.

Una pregunta que de inmediato se viene a la mente es que si *m* no es entregado a ningún proceso, ¿cómo se puede hablar de un protocolo de multitransmisión *confiable*? En principio, existe sólo un caso en el que a la entrega de *m* se le permite fallar: cuando el cambio de la membresía del grupo es resultado de la congelación del remitente de *m*. En ese caso, o todos los miembros de *G* deberán enterarse del aborto del nuevo miembro, o ninguno. Alternativamente, *m* puede ser ignorado por cada miembro, lo cual corresponde a la situación en la que el remitente se congeló antes de enviar *m*.

Esta forma más potente de multitransmisión confiable garantiza que un mensaje multitransmitido a la vista de grupo *G* sea entregado a cada proceso no defectuoso incluido en *G*. Si el remitente del mensaje se congela durante la multitransmisión, el mensaje puede o ser entregado a todos los procesos restantes o ignorado por cada uno de ellos. Se dice que una multitransmisión confiable con esta propiedad es **virtualmente sincrónica** (Birman y Joseph, 1987).

Consideremos los cuatro procesos mostrados en la figura 8-13. En cierto punto en el tiempo, el proceso P_1 se une al grupo, el cual entonces se compone de P_1, P_2, P_3 y P_4. Después de que algunos mensajes han sido multitransmitidos, P_3 se congela. Sin embargo, antes de congelarse multitransmitió con éxito un mensaje hacia los procesos P_2 y P_4, pero no a P_1. No obstante, la sincronía virtual garantiza que el mensaje no sea entregado en absoluto, estableciendo efectivamente la situación de que el mensaje nunca fue enviado antes de que P_3 se congelara.

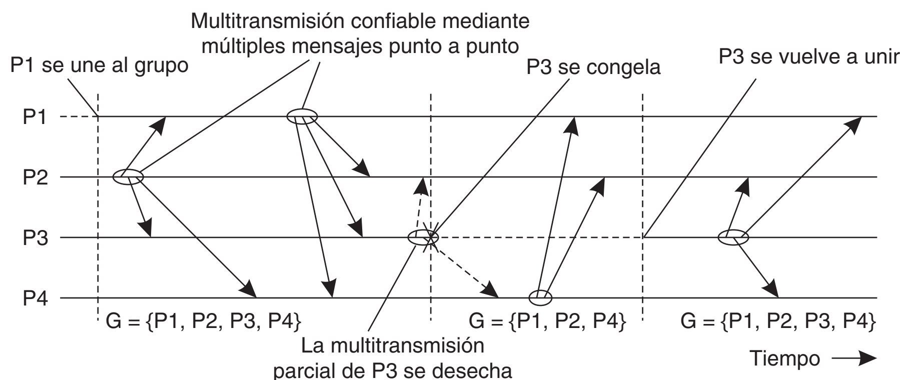


Figura 8-13. Principio de multitransmisiones síncronas virtuales.

Una vez que P_3 ha sido eliminado del grupo, la comunicación prosigue entre los miembros restantes. Posteriormente, cuando P_3 se recupera, puede unirse al grupo otra vez en cuanto su estado se actualiza.

El principio de sincronía virtual se deriva del hecho de que todas las multitransmisiones ocurren entre cambios de vista. Expresado de otra manera, un cambio de vista actúa como una barrera a través de la cual no puede pasar ninguna multitransmisión. En cierto sentido, es comparable al uso de una variable de sincronización en almacenamientos de datos distribuidos como se vio en el capítulo anterior. Todas las multitransmisiones que están en tránsito mientras ocurre un cambio de vista se completan antes de que el cambio de vista entre en vigor. La implementación de sincronía virtual no es trivial, como se verá con todo detalle más adelante.

Ordenamiento de los mensajes

La sincronía virtual permite a un desarrollador de aplicaciones considerar que las multitransmisiones ocurren en épocas separadas por cambios de membresía al grupo. Sin embargo, aún no se ha dicho nada sobre el ordenamiento de las multitransmisiones. En general, se distinguen cuatro ordenamientos diferentes.

1. Multitransmisiones no ordenadas
2. Multitransmisiones ordenadas en modo FIFO (primera en llegar, primera en salir)
3. Multitransmisiones causalmente ordenadas
4. Multitransmisiones totalmente ordenadas

Una **multitransmisión no ordenada confiable** es una multitransmisión virtualmente síncrona en la cual no existen garantías en cuanto al orden en el que los mensajes recibidos son entregados por los diferentes procesos. Para explicarlo, supongamos que una biblioteca que envía y recibe soporta la multitransmisión confiable. La operación de recepción bloquea el proceso de invocación hasta que recibe un mensaje.

Proceso 1	Proceso 2	Proceso 3
envía m1	recibe m1	recibe m2
envía m2	recibe m2	recibe m1

Figura 8-14. Tres procesos en comunicación en el mismo grupo. El orden de los eventos por proceso se muestra a lo largo del eje vertical.

Supongamos ahora que un remitente P_1 multitransmite dos mensajes a un grupo en tanto que otros dos procesos de éste esperan la llegada de mensajes, como se muestra en la figura 8-14. Suponiendo que los procesos no se congelan o abandonan el grupo durante estas multitransmisiones, es posible que la capa de comunicación ubicada en P_2 reciba primero el mensaje m_1 y luego m_2. Como no existen restricciones en cuanto al orden de llegada de los mensajes, éstos pueden ser entregados a P_2 en el orden en que se reciben. Por contraste, la capa de comunicación localizada en P_3 puede recibir primero el mensaje m_2 seguido por m_1 y entregarlos en el mismo orden a P_3.

En el caso de **multitransmisiones confiables ordenadas en modo FIFO**, la capa de comunicación es obligada a entregar los mensajes que llegan del mismo proceso en el mismo orden en que fueron enviados. Consideremos la comunicación dentro de un grupo de cuatro procesos, como se muestra en la figura 8-15. Con ordenamiento FIFO, lo único importante es que el mensaje m_1 siempre sea entregado antes que m_2, y, asimismo, que el mensaje m_3 siempre sea entregado antes que m_4. Esta regla tiene que ser obedecida por todos los procesos presentes en el grupo. En otros términos, cuando la capa de comunicación ubicada en P_3 recibe m_2 primero, esperará para entregarlo a P_3 hasta que haya recibido y entregado m_1.

Proceso 1	Proceso 2	Proceso 3	Proceso 4
envía m1	recibe m1	recibe m3	envía m3
envía m2	recibe m3	recibe m1	envía m4
	recibe m2	recibe m2	
	recibe m4	recibe m4	

Figura 8-15. Cuatro procesos en el mismo grupo con dos remitentes distintos, y un posible orden de entrega de los mensajes en la multitransmisión en orden FIFO.

Sin embargo, no existe ninguna restricción en cuanto a la entrega de mensajes enviados por diferentes procesos. En otros términos, si P_2 recibe m_1 antes que m_3, puede entregar los dos mensajes en ese orden. Entre tanto, puede ser que el proceso P_3 haya recibido m_3 antes que m_1. El ordenamiento FIFO establece que P_3 puede entregar m_3 antes que m_1, aunque este orden de entrega es diferente del de P_2.

Por último, la **multitransmisión confiable causalmente ordenada** entrega mensajes de tal forma que se preserve la causalidad entre los diferentes mensajes. En otros términos, si un mensaje m_1 causalmente precede a otro mensaje m_2, sin importar si fueron multitransmitidos por el mismo remitente, entonces la capa de comunicación localizada en cada destinatario siempre entregará m_2 una vez que haya recibido y entregado m_1. Observe que las multitransmisiones causalmente ordenadas pueden ser implementadas mediante marcas de tiempo vectoriales como se vio en el capítulo 6.

Aparte de estos tres ordenamientos, puede haber la restricción adicional de que la entrega de mensajes también se haga en una forma totalmente ordenada. **Entrega totalmente ordenada** significa que aunque la entrega de los mensajes se haga sin orden, en orden FIFO, o en orden causal, adicionalmente se requiere que cuando los mensajes sean entregados, se entreguen en el mismo orden a todos los miembros del grupo.

Por ejemplo, con la combinación de multitransmisión en total orden y en orden FIFO, los procesos P_2 y P_3 ilustrados en la figura 8-15 pueden entregar primero el mensaje m_3 y luego el mensaje m_1. Sin embargo, si P_2 entrega m_1 antes que m_3, en tanto que P_3 entrega m_3 antes de entregar m_1, violarían la restricción de total ordenamiento. Observemos que el ordenamiento FIFO deberá seguirse respetando. En otros términos, m_2 deberá ser entregado después de m_1 y, del mismo modo, m_4 deberá ser entregado después de m_3.

La multitransmisión confiable sincrónica que ofrece entrega de mensajes en total orden se conoce virtualmente como **multitransmisión atómica**. Con las tres diferentes restricciones de ordenamiento de mensajes previamente analizadas, se llega a seis formas de multitransmisión confiable como se muestra en la figura 8-16 (Hadzilacos y Toueg, 1993).

Implementación de sincronía virtual

A continuación consideramos una posible implementación de una multitransmisión confiable virtualmente síncrona. Un ejemplo de ésta aparece en Isis, un sistema distribuido tolerante a fallas que ha estado en uso práctico en la industria durante varios años. Nos enfocaremos en algunos

Multitransmisión	Ordenamiento básico de los mensajes	¿Entrega en orden total?
Multitransmisión confiable	Ninguno	No
Multitransmisión FIFO	Entrega en orden FIFO	No
Multitransmisión causal	Entrega en orden causal	No
Multitransmisión atómica	Ninguno	Sí
Multitransmisión atómica FIFO	Entrega en orden FIFO	Sí
Multitransmisión atómica causal	Entrega en orden causal	Sí

Figura 8-16. Seis versiones distintas de multitransmisión confiable virtualmente síncrona.

de los temas de implementación de esta técnica en la forma que describen Birman y colaboradores (1991).

En Isis, la multitransmisión confiable utiliza las funciones de comunicación punto a punto confiables disponibles de la red subyacente, en particular, TCP. La multitransmisión de un mensaje *m* a un grupo de procesos se implementa mediante el envío confiable de *m* a cada miembro del grupo. En consecuencia, aunque está garantizado que suceda cada transmisión, no existe garantía alguna de que *todos* los miembros del grupo reciban *m*. En particular, el remitente puede fallar antes de haber transmitido *m* a cada miembro.

Además de comunicación confiable punto a punto, Isis también asume que los mensajes de la misma fuente son recibidos por una capa de comunicación en el orden en que fueron enviados por dicha fuente. En la práctica, este requerimiento se resuelve por medio de conexiones TCP para comunicación punto a punto.

El problema principal a resolver es garantizar que todos los mensajes enviados a la vista *G* sean entregados a todos los procesos no defectuosos presentes en *G* antes de que ocurra el siguiente cambio de membresía al grupo. El primer tema a ser atendido es asegurarse de que cada proceso presente en *G* haya recibido todos los mensajes enviados a *G*. Observemos que como el remitente de un mensaje *m* a *G* puede haber fallado antes de completar su multitransmisión, si puede haber procesos en *G* que nunca recibirán *m*. Como el remitente se congeló, estos procesos deberán obtener *m* de alguna otra parte. Cómo detectar un proceso que no ha recibido un mensaje se explica a continuación.

La solución a este problema es permitir que cada proceso incluido en *G* mantenga *m* hasta asegurarse de que todos los miembros de *G* lo hayan recibido. Se dice que *m* es **estable** si todos los miembros presentes en *G* lo recibieron. Se permite que sólo sean entregados los mensajes estables. Para garantizar su estabilidad, es suficiente con seleccionar un proceso arbitrario (operacional) en *G* y solicitarle que envíe *m* a todos los demás procesos.

Para ser más específicos, supongamos que la vista actual es G_i, pero que se necesita instalar la siguiente vista G_{i+1}. Sin que se pierda la generalidad, podemos suponer que G_i y G_{i+1} difieren cuando mucho en un proceso. Un proceso *P* notifica el cambio de vista cuando recibe un mensaje de cambio de vista. El mensaje puede provenir del proceso que desea unirse al grupo o abandonarlo, o de un proceso que había detectado la falla de un proceso en G_1 que ahora debe ser eliminado, como se muestra en la figura 8-17(a).

Cuando un proceso *P* recibe el mensaje de cambio de vista a G_{i+1}, primero remite una copia de cualquier mensaje inestable de G_i que aún tenga a cada proceso incluido en G_{i+1} y posteriormente lo marca como estable. Recordemos que Isis supone que la comunicación punto a punto es confiable, de tal forma que los mensajes remitidos nunca se pierden. Dicha operación de remisión garantiza que todos los mensajes presentes en G_i que han sido recibidos por cuando menos un proceso sean recibidos por todos los procesos no defectuosos de G_i. Observe que también habría sido suficiente con seleccionar un coordinador único para remitir mensajes inestables.

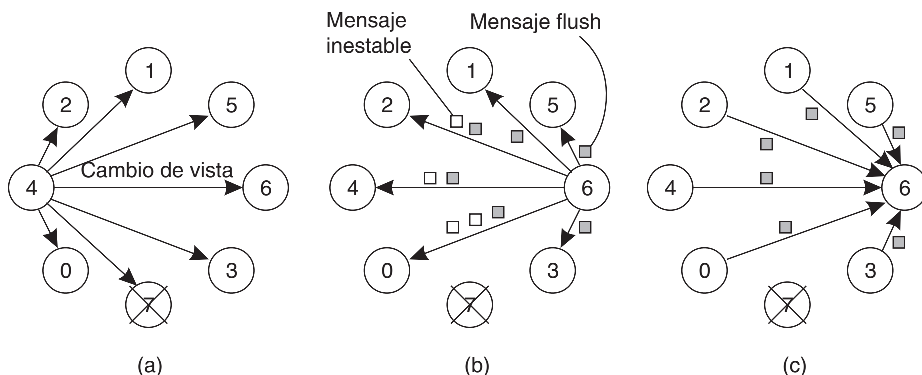


Figura 8-17. (a) El proceso 4 advierte que el proceso 7 se congeló y envía un cambio de vista. (b) El proceso 6 envía todos sus mensajes inestables, seguidos por un mensaje flush. (c) El proceso 6 instala la nueva vista cuando ha recibido un mensaje flush de todos los demás.

Para indicar que *P* ya no tiene ningún mensaje inestable y que está preparado para instalar G_{i+1} en cuanto los demás procesos también puedan hacerlo, multitransmite un **mensaje flush** para G_{i+1}, como se muestra en la figura 8-17(b). Después de que *P* ha recibido un mensaje flush para informar sobre G_{i+1} a partir de cada uno de los demás procesos, puede instalar con seguridad la nueva vista [mostrada en la figura 8-17(c)].

Cuando un proceso *Q* recibe un mensaje *m* enviado en G_i, y *Q* continúa creyendo que la vista actual es G_i, entrega *m* sin tomar en cuenta ningunas restricciones adicionales de ordenamiento de los mensajes. Si ya había recibido *m*, considera el mensaje como un duplicado y además lo desecha.

Como el proceso *Q* recibirá con el tiempo el mensaje de cambio de vista para G_{i+1}, también remitirá primero cualquiera de sus mensajes inestables y posteriormente concluirá las cosas al enviar un mensaje flush para G_{i+1}. Observe que debido al orden de entrega de los mensajes en la capa de comunicación, siempre se recibe un mensaje flush de un proceso después de recibir un mensaje inestable de ese mismo proceso.

El principal defecto del protocolo descrito hasta ahora es que no puede ocuparse de fallas de proceso mientras se está anunciando un nuevo cambio de vista. En particular, supone que hasta que

la nueva vista G_{i+1} ha sido instalada por cada miembro del grupo incluido en G_{i+1}, ningún proceso G_{i+1} fallará (lo cual conduce a la siguiente vista G_{i+2}). El problema se resuelve anunciando cambios de vista para cualquier G_{i+k} incluso mientras los cambios previos aún no han sido instalados por todos los procesos. Los detalles se dejan como ejercicio para el lector.

8.5 REALIZACIÓN DISTRIBUIDA

El problema de multitransmisión atómica analizado en la sección previa es ejemplo de un problema todavía más general, conocido como **realización distribuida**. El problema de realización distribuida implica lograr que una operación sea realizada por cada miembro de un grupo o por ninguno en absoluto. En el caso de multitransmisión confiable, la operación es la entrega de un mensaje. Con transacciones distribuidas, la operación puede ser la realización de una transacción en un solo sitio que interviene en la transacción. Otros ejemplos de realización distribuida y cómo puede ser resuelta se analizan en Tanisch (2000).

La realización distribuida a menudo se establece por medio de un coordinador. En un esquema simple, este coordinador comunica a todos los demás procesos involucrados, llamados participantes, si deben realizar o no (localmente) la operación de que se trate. Este esquema se refiere a un **protocolo de realización monofásico**, y tiene la desventaja evidente de que si uno de los participantes en efecto no puede realizar la operación, no hay forma de comunicárselo al coordinador. Por ejemplo, en el caso de transacciones distribuidas, no es posible implementar una realización local porque violaría las restricciones de control de concurrencia.

En la práctica se requieren esquemas más complejos, de los cuales el más común es el protocolo de realización bifásico que vamos a estudiar con todo detalle a continuación. El defecto principal de este protocolo es que no puede manejar con eficiencia la falla del coordinador. Con esa finalidad, se ha desarrollado un protocolo trifásico, el cual también presentamos.

8.5.1 Realización bifásica

El **protocolo de realización bifásico** (**2PC**, del inglés *two-phase commit protocol*) se debe a Gray (1978). Sin que se pierda la generalidad, consideremos una transacción distribuida que implica la participación de varios procesos y donde cada proceso se ejecuta en una máquina diferente. Suponiendo que no ocurren fallas, el protocolo se compone de las dos fases siguientes, cada fase comprende dos pasos [vea también Bernstein y cols. (1987)]:

1. El coordinador envía un mensaje *VOTE_REQUEST* a todos los participantes.

2. Cuando un participante recibe un mensaje *VOTE_REQUEST*, regresa el mensaje *VOTE_COMMIT* al coordinador para decirle que se prepare para realizar localmente su parte de la transacción, o de lo contrario envía un mensaje *VOTE_ABORT*.

3. El coordinador reúne todos los votos de los participantes. Si todos los participantes votaron para realizar la transacción, entonces también lo hará el coordinador. En ese caso envía un mensaje *GLOBAL_COMMIT* a todos los participantes. Sin embargo, si un participante ha votado abortar la transacción, el coordinador también decidirá abortar la transacción y multitransmitir el mensaje *GLOBAL_ABORT*.

4. Cada participante que votó por la realización espera la reacción final del coordinador. Si un participante recibe un mensaje *GLOBAL_COMMIT*, entonces realiza localmente la transacción. De lo contrario, cuando se recibe un mensaje *GLOBAL_ABORT*, la transacción también es abortada localmente.

La primera fase es la fase de votación, y consiste en los pasos 1 y 2. La segunda es la fase de decisión y consiste en los pasos 3 y 4. Estos cuatro pasos se muestran como diagramas de estado finito en la figura 8-18.

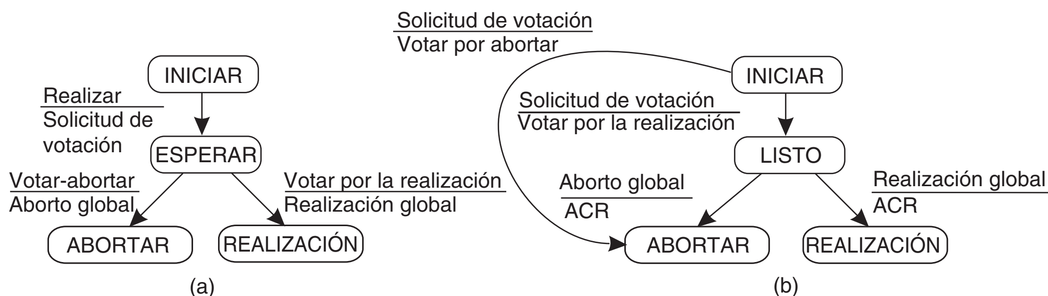


Figura 8-18. (a) Máquina de estado finito para el coordinador en 2PC, (b) Máquina de estado finito para un participante.

Surgen varios problemas cuando se utiliza este protocolo 2PC básico en un sistema que presenta fallas. En primer lugar, observe que tanto el coordinador como los participantes tienen estados en los que se bloquean esperando los mensajes entrantes. Por consiguiente, el protocolo puede fallar con facilidad cuando un proceso se congela, puesto que otros procesos pueden quedarse esperando por tiempo indefinido la llegada de un mensaje enviado por dicho proceso. Por esta razón se utilizan mecanismos de tiempo fuera. Estos mecanismos se explican en las siguientes páginas.

Cuando se examinan las máquinas de estado finito ilustradas en la figura 8-18, se puede ver que existen tres estados en los que o un coordinador o un participante se bloquea en espera de un mensaje entrante. En primer lugar, un participante puede estar esperando en su estado *INIT* a que el coordinador envíe un mensaje *VOTE_REQUEST*. Si no lo recibe después de cierto lapso de tiempo, el participante simplemente decidirá abortar localmente la transacción y, por tanto, enviará un mensaje *VOTE_ABORT* al coordinador.

Asimismo, el coordinador puede estar bloqueado en el estado *WAIT*, en espera de los votos de cada participante. Si no todos los votos han sido recolectados después de cierto lapso de tiempo, el

coordinador también deberá votar por el aborto, y posteriormente enviar un mensaje *GLOBAL_ABORT* a todos los participantes.

Por último, un participante puede estar bloqueado también en el estado *READY*, en espera del voto global enviado por el coordinador. Si no se recibe ese mensaje dentro de un tiempo dado, el participante simplemente no puede decidir abortar la transacción. En vez de eso, debe indagar qué mensaje envió realmente el coordinador. La solución más simple a este problema es permitir que cada participante se bloquee hasta que el coordinador se recupere de nuevo.

Una solución mejor es permitir que un participante *P* se ponga en contacto con otro participante *Q* para ver si éste puede decidir a partir del estado actual de *Q* lo que deberá hacerse. Por ejemplo, supongamos que *Q* había llegado al estado *COMMIT*. Esto es posible sólo si el coordinador había enviado un mensaje *GLOBAL_COMMIT* a *Q* justo antes de congelarse. Aparentemente, este mensaje aún no había sido enviado a *P*. Por consiguiente, *P* ahora también puede decidir realizar localmente. Asimismo, si *Q* se encuentra en el estado *ABORT*, *P* también puede abortar con seguridad.

Supongamos ahora que *Q* aún está en el estado *INIT*. Esta situación puede ocurrir cuando el coordinador ha enviado un mensaje *VOTE_REQUEST* a todos los participantes y este mensaje ha llegado a *P* (el cual respondió enseguida con el mensaje *VOTE_COMMIT*), pero no ha llegado a *Q*. En otros términos, el coordinador se habría congelado mientras multitransmitía el mensaje *VOTE_REQUEST*. En este caso, es seguro abortar la transacción: tanto *P* como *Q* pueden llevar una transición al estado *ABORT*.

La situación más difícil ocurre cuando *Q* también se encuentra en el estado *READY*, en espera de una respuesta del coordinador. En particular, si todos los participantes se encuentran en el estado *READY*, no se puede tomar una decisión. El problema es que aunque todos los participantes desean realizar, siguen necesitando el voto del coordinador para llegar a la decisión final. Por consiguiente, el protocolo se bloquea hasta que el coordinador se recupera.

Las diversas alternativas se resumen en la figura 8-19.

Estado de Q	**Acción por P**
REALIZAR	Hacer transición a REALIZAR
ABORTAR	Hacer transición a ABORTAR
INICIAR	Hacer transición a ABORTAR
LISTO	Contactar a otro participante

Figura 8-19. Acciones tomadas por un participante *P* cuando se encuentra en el estado *READY* (LISTO) y ha contactado con otro participante *Q*.

Para garantizar que un proceso pueda recuperarse en realidad, es necesario que conserve su estado en almacenamiento persistente. (Cómo se pueden guardar datos en una forma tolerante a las fallas se analiza más adelante en este capítulo.) Por ejemplo, si un participante se encontraba en el estado *INIT*, al recuperarse puede decidir con seguridad abortar localmente la transacción y luego

informar al coordinador. Asimismo, cuando ya había tomado una decisión tal como congelarse mientras se encontraba en el estado *COMMIT* o *ABORT*, es para regresar a dicho estado y retransmitir su decisión al coordinador.

Surgen problemas cuando un participante se congela mientras se encuentra en el estado *READY*. En ese caso, cuando se recupera, no puede decidir por sí mismo lo que debe hacer a continuación, es decir, realizar o abortar la transacción. En consecuencia, se ve obligado a ponerse en contacto con otros participantes para indagar qué debe hacer, análogo a la situación en que se pone fuera de servicio mientras reside en el estado *READY* como ya se describió.

El coordinador tiene sólo dos estados críticos de los que tiene que estar al tanto. Cuando inicia el protocolo 2PC, deberá indicar que está entrando al estado *WAIT* de modo que posiblemente pueda retransmitir el mensaje *VOTE_REQUEST* a todos los participantes después de recuperarse. Asimismo, si había tomado una decisión en la segunda fase, es suficiente si dicha decisión ha sido registrada de modo que pueda ser retransmitida cuando se recupere.

En la figura 8-20 proporcionamos un plan general de las acciones ejecutadas por el coordinador. Éste primero multitransmite un mensaje *VOTE_REQUEST* a todos los participantes para recopilar su votos; posteriormente registra que está entrando al estado *WAIT*, después de lo cual espera la llegada de los votos de los participantes.

Acciones realizadas por el coordinador

```
escribir START_2PC en el registro local;
multitransmitir VOTE_REQUEST a todos los participantes;
en tanto no se reciban todos los votos {
    esperar por cualquier voto entrante;
    si llegan fuera de tiempo {
        escribir GLOBAL_ABORT en el registro local;
        multitransmitir GLOBAL_ABORT a todos los participantes;
        salir;
    }
    registrar voto;
}
si todos los participantes enviaron VOTE_COMMIT y el coordinador vota por COMMIT {
    escribir GLOBAL_COMMIT en el registro local;
    multidifusionar GLOBAL_COMMIT a todos los participantes;
} sino {
    escribir GLOBAL_ABORT en el registro local;
    multitransmitir GLOBAL_ABORT a todos los participantes;
}
```

Figura 8-20. Bosquejo de los pasos seguidos por el coordinador en un protocolo de realización bifásico.

Si no todos los votos han sido recolectados pero no se reciben más votos dentro de un intervalo de tiempo dado prescrito de antemano, el coordinador asume que uno o más participantes han fallado. Por consiguiente, deberá abortar la transacción y multitransmitir un mensaje *GLOBAL_ABORT* a los participantes (restantes).

Si no ocurren fallas, a la larga el coordinador habrá recopilado todos los votos. Si todos los participantes tanto como el coordinador votan por la realización, primero se registra un mensaje *GLOBAL_COMMIT* y posteriormente se envía a todos los procesos. De lo contrario, el coordinador multitransmite un mensaje *GLOBAL_ABORT* (después de registrarlo en el registro local).

La figura 8-21(a) muestra los pasos seguidos por un participante. En primer lugar, el proceso espera a que el coordinador le solicite votar. Observe que esta espera puede hacerse por un hilo distinto que se ejecuta en el espacio de la dirección del proceso. Si no llega ningún mensaje, la transacción simplemente es abortada. Aparentemente, el coordinador ha fallado.

Después de recibir una solicitud de votación, el participante puede decidir votar por la realización de la transición, para ello registra primero su decisión en un registro local y luego informa al coordinador enviando un mensaje *VOTE_COMMIT*. El participante debe esperar entonces la decisión global. Asumiendo que esta decisión (la cual otra vez deberá venir del coordinador) llega a tiempo, simplemente se registra en el registro local, tras de lo cual puede ser realizada.

Sin embargo, cuando un participante queda fuera de servicio mientras llega la decisión del coordinador, ejecuta un protocolo de terminación multitransmitiendo primero un mensaje *DECISION_REQUEST* a los demás procesos, después de lo cual se bloquea en espera de una respuesta. Cuando llega una respuesta (posiblemente del coordinador, el cual se supone que con el tiempo se recuperó), el participante anota la decisión en su registro local y la maneja como corresponde.

Cada participante deberá estar preparado para aceptar solicitudes para una decisión global de los demás participantes. Con ese fin, supongamos que cada participante inicia un hilo distinto, ejecutando concurrentemente con el hilo principal del participante como se muestra en la figura 8-21(b). Este hilo se bloquea hasta que recibe una solicitud de decisión. Sólo puede servir de ayuda a otro proceso si su participante asociado ya llegó a una decisión final. En otros términos, si se había escrito un mensaje *GLOBAL_COMMIT* o *GLOBAL_ABORT* en el registro local, es seguro que el coordinador había enviado por lo menos su decisión a este proceso. Además, el hilo también puede decidir enviar un mensaje *GLOBAL_ABORT* cuando su participante asociado aún está en el estado *INIT*, como previamente se analizó. En todos los demás casos, el hilo receptor no puede ayudar y el participante solicitante no recibirá una respuesta.

Lo que se observa es que puede ser posible que un participante tenga que bloquearse hasta que el coordinador se recupere. Esta situación ocurre cuando todos los participantes han recibido y procesado el mensaje *VOTE_REQUEST* del coordinador, mientras que en el ínterin el coordinador se congela. En ese caso, los participantes no pueden decidir cooperativamente sobre la acción final que deberá tomarse. Por esta razón, el protocolo 2PC también se conoce como **protocolo de realización de bloqueo**.

Existen varias soluciones para evitar el bloqueo. Una, descrita por Babaoglu y Toueg (1993), es utilizar un primitivo de multitransmisión mediante el cual un destinatario multitransmite de inmediato un mensaje recibido a todos los demás procesos. Se puede demostrar que este método permite que un participante llegue a una decisión final, aun cuando el coordinador todavía no se haya recuperado. Otra solución es el protocolo de realización trifásica, el cual es el tema de esta sección y lo abordamos enseguida.

```
Acciones emprendidas por el participante:

    escribir INIT en el registro local;
    esperar mensaje VOTE_REQUEST enviado por el coordinador;
    si llega fuera de tiempo {
        escribir VOTE_ABORT en el registro local;
        salir;
    }
    si el participante vota por COMMIT {
        escribir VOTE_COMMIT en el registro local;
        enviar VOTE_COMMIT al coordinador;
        esperar DECISION enviado por el coordinador;
        si llega fuera de tiempo {
            multitransmitir DECISION_REQUEST a otros participantes;
            esperar hasta recibir DECISION; /* permanecer bloqueado */
            escribir DECISION en el registro local;
        }
        si DECISION == GLOBAL_COMMIT
            escribir GLOBAL_COMMIT en el registro local;
        sino si DECISION == GLOBAL_ABORT
            escribir GLOBAL_ABORT en el registro local;
    } sino }
        escribir VOTE_ABORT en el registro local;
        enviar VOTE_ABORT al coordinador;
    }
```

(a)

```
Acciones para manejar solicitudes de decisión: /* ejecutadas por un hilo distinto */

    mientras sea verdadero {
        esperar hasta que se reciba cualquier DECISION_REQUEST entrante; /* permanecer bloqueado */
        leer STATE (ESTADO) más recientemente registrado en el registro local;
        si STATE == GLOBAL_COMMIT
            enviar GLOBAL_COMMIT a participante solicitante;
        sino si STATE == INIT or STATE == GLOBAL_ABORT
            enviar GLOBAL_ABORT a participante solicitante;
        sino
            salto; /* participante permanece bloqueado */
    }
```

(b)

Figura 8-21. a) Pasos seguidos por un proceso participante en 2PC. (b) Pasos para manejar las solicitudes entrantes de tomar una decisión.

8.5.2 Realización trifásica

Un problema con el protocolo de realización bifásica es que cuando el coordinador se congela, es posible que los participantes no puedan llegar a una decisión final. En consecuencia, es posible que los participantes deban permanecer bloqueados hasta que el coordinador se recupere. Skeen (1981) desarrolló una variante del 2PC, llamada **protocolo de realización trifásica** (**3PC**, del inglés *three-phase commit protocol*), que impide a los procesos bloquearse ante la presencia de congelaciones por detención. Aunque el 3PC es ampliamente referido en la literatura, en la práctica no se aplica a menudo ya que las condiciones en que el 2PC se bloquea rara vez ocurren. El protocolo se estudia aquí porque da una idea más completa de cómo resolver problemas de tolerancia a fallas en sistemas distribuidos.

Al igual que el 2PC, el 3PC también está formulado en función de un coordinador y varios participantes. Sus respectivas máquinas de estado finito se muestran en la figura 8-22. La esencia de este protocolo es que los estados del coordinador y de cada participante satisfacen las dos condiciones siguientes:

1. No existe un estado único a partir del cual sea posible realizar una transición directamente ante un estado *COMMIT* o un estado *ABORT*.
2. No existe ningún estado en el cual no sea posible elaborar una conclusión final, y a partir de la cual se pueda realizar una transición ante un estado *COMMIT*.

Se puede demostrar que estas dos condiciones son necesarias y suficientes para que un protocolo de realización no se bloquee (Skeen y Stonebraker, 1983).

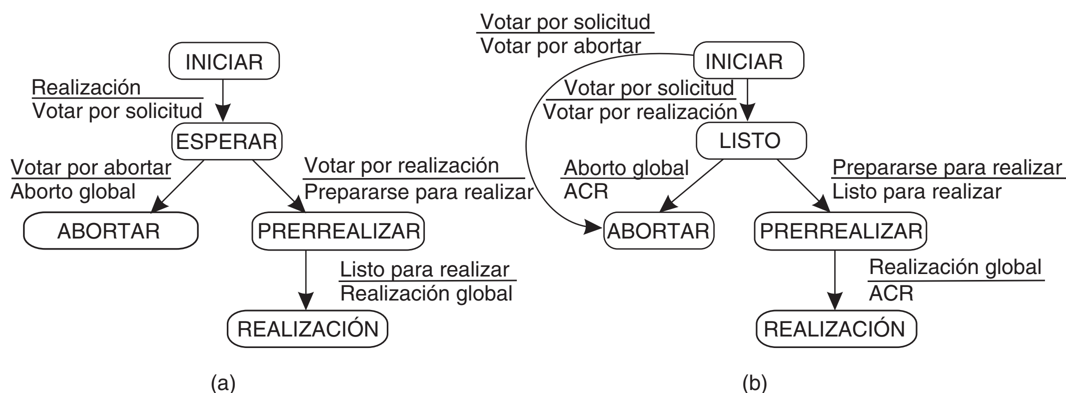


Figura 8-22. (a) Máquina de estado finito para el coordinador en 3PC. (b) Máquina de estado finito para un participante.

El coordinador en 3PC primero envía un mensaje *VOTE_REQUEST* a todos los participantes, después de lo cual espera las respuestas. Si cualquier participante vota por abortar la transacción, la decisión final también será abortar, por lo que el coordinador envía un mensaje *GLOBAL_ABORT*.

No obstante, cuando la transacción puede ser realizada, se envía un mensaje *PREPARE_COMMIT*. Sólo después de que cada participante haya confirmado su recepción estará preparado para realizar, el coordinador enviará el mensaje final *GLOBAL_COMMIT* mediante el cual la transacción es en efecto realizada.

De nuevo, existen sólo algunas situaciones en las que un proceso se bloquea mientras espera la llegada de mensajes. En primer lugar, si un participante está esperando la solicitud de voto del coordinador mientras se encuentra en el estado *INIT*, finalmente cambiará al estado *ABORT*, al suponer que el coordinador se congeló. Esta situación es idéntica a la implementada en 2PC. Análogamente, el coordinador puede estar en el estado *WAIT*, en espera de los votos de los participantes. En un tiempo fuera, el coordinador concluirá que un participante se congeló y, por tanto, abortará la transacción multitransmitiendo un mensaje *GLOBAL_ABORT*.

Supongamos ahora que el coordinador está bloqueado en el estado *PRECOMMIT*. En un tiempo fuera de servicio, concluirá que uno de los participantes se había congelado, pero se sabe que dicho participante votó por la realización de la transacción. Por consiguiente, el coordinador puede instruir con seguridad a los participantes operacionales para que efectúen realización multitransmitiendo un mensaje *GLOBAL_COMMIT*. Además, confía en un protocolo de recuperación para que el participante congelado finalmente realice su parte de la transacción cuando se recupere.

Un participante *P* puede bloquearse en el estado *READY* o en el estado *PRECOMMIT*. En un tiempo fuera de servicio, *P* puede concluir sólo que el coordinador ha fallado, de modo que entonces debe indagar qué hacer a continuación. Como en el protocolo 2PC, si *P* se pone en contacto con cualquier otro participante que se encuentre en el estado *COMMIT* (o *ABORT*), también deberá pasarse a ese estado. Además, si todos los participantes se encuentran en el estado *PRECOMMIT*, la transacción puede ser realizada con seguridad.

De nuevo como en el 2PC, si otro participante *Q* aún está en el estado *INIT*, la transacción puede ser abortada con seguridad. Es importante señalar que *Q* puede estar en el estado *INIT* sólo si ningún otro participante se encuentra en el estado *PRECOMMIT*. Un participante puede llegar a *PRECOMMIT* sólo si el coordinador había alcanzado el estado *PRECOMMIT* antes de congelarse y, por tanto, ha recibido un voto por la realización de cada participante. En otros términos, ningún participante puede residir en el estado *INIT* mientras otro participante se encuentre en el estado *PRECOMMIT*.

Si cada uno de los participantes con los que *P* puede contactarse se encuentra en el estado *READY* (y juntos forman una mayoría), la transacción deberá ser abortada. El punto a destacar es que otro participante puede haberse congelado y que posteriormente se recuperará. Sin embargo, ni *P* ni ningún otro participante operacional sabe cuál será el estado del participante congelado cuando se recupere. Si el proceso recupera el estado *INIT*, entonces decidir abortar la transacción es la única decisión correcta. En el peor de los casos, el proceso puede recuperar el estado *PRECOMMIT*, pero entonces no pasa nada si se aborta la transacción.

Esta situación es la diferencia principal con el 2PC, donde un participante congelado podría recuperar el estado *COMMIT* mientras todos los demás procesos continúan en el estado *READY*. En ese caso, los procesos operacionales restantes no podrían llegar a una conclusión final y tendrían que esperar hasta que el proceso congelado se recupere. Con el 3PC, si cualquier proceso operacional se encuentra en el estado *READY*, ningún proceso congelado se recuperará a un estado distinto

de *INIT*, *ABORT* o *PRECOMMIT*. Por esta razón, los procesos sobrevivientes siempre llegan a una decisión final.

Por último, si los procesos que *P* puede alcanzar se encuentran en el estado *PRECOMMIT* (y forman una mayoría), entonces es seguro realizar la transacción. De nueva cuenta, se puede demostrar que, en este caso, todos los demás procesos estarán o en el estado *READY* o, por lo menos, recuperarán el estado *READY*, *PRECOMMIT* o *COMMIT* que tenían cuando se habían congelado.

Más detalles sobre el 3PC pueden encontrarse en Bernstein y colaboradores (1987), y en Chow y Johnson (1997).

8.6 RECUPERACIÓN

Hasta ahora, hemos abordado principalmente algoritmos que permiten tolerar fallas. Sin embargo, una vez ocurrida una falla, resulta esencial que el proceso donde aconteció pueda recuperarse a un estado correcto. En adelante, primero abordamos lo que en realidad significa recuperarse a un estado correcto, y posteriormente vemos cuándo y cómo el estado de un sistema distribuido puede registrarse y recuperarse, por medio de marcación de puntos de control y registro de mensajes.

8.6.1 Introducción

La recuperación de errores es fundamental para la tolerancia a fallas. Recordemos que un error es esa parte de un sistema que puede conducir a una falla. La idea integral sobre recuperación de errores es reemplazar un estado erróneo con un estado libre de error. Esencialmente, existen dos formas de recuperación de errores.

En la **recuperación hacia atrás**, lo principal es hacer que el sistema regrese de su estado actual erróneo a su estado previamente correcto. Para lograrlo, será necesario registrar el estado del sistema de vez en cuando y, cuando las cosas vayan mal, restaurar el estado registrado. Cada vez que se registra (una parte de) el estado actual del sistema, se dice que se **marca un punto de control**.

Otra forma de recuperación de errores es la **recuperación hacia adelante**. En este caso, cuando el sistema ha entrado a un estado erróneo, en lugar de regresarlo a un estado de punto de control previo, se intenta llevarlo a un nuevo estado correcto a partir del cual se pueda continuar ejecutando. El problema principal con los mecanismos de recuperación de errores hacia adelante es que se debe saber de antemano qué errores pueden ocurrir. Sólo en ese caso es posible corregir los errores y trasladarse a un nuevo estado.

La distinción entre la recuperación de errores hacia atrás y hacia adelante se explica con facilidad cuando consideramos la implementación de comunicación confiable. El método común de recuperar un paquete perdido es permitir que el remitente retransmita el paquete. En efecto, la retransmisión de paquetes establece que intentamos regresar a un estado correcto previo, es decir, a aquél en el que se perdió el paquete que está siendo enviado. La comunicación confiable a través

de la retransmisión de paquetes es, por consiguiente, un ejemplo de la aplicación de técnicas de recuperación de errores hacia atrás.

Un procedimiento alternativo es utilizar un método conocido como **corrección por borradura**. En este procedimiento, se construye un paquete perdido a partir de otro para entregar los paquetes con éxito. Por ejemplo, en un código de borradura de bloque (n,k), un conjunto de *k paquetes originales* se cifra en la forma de *n paquetes cifrados*, de modo que basta *un* juego de k paquetes cifrados para reconstruir los k paquetes originales. Valores típicos son $k = 16$ o $k = 32$, y $k < n \leq 2k$ [vea, por ejemplo, Rizzo (1997)]. Si aún no se han entregado suficientes paquetes, el remitente deberá continuar transmitiendo paquetes hasta que un paquete previamente perdido pueda ser construido. La corrección por borradura es un ejemplo típico de un método de recuperación de errores hacia adelante.

En sistemas distribuidos, por mucho, las técnicas de recuperación de errores hacia atrás son las más aplicadas como mecanismo general para recuperarse de las fallas. El beneficio principal de la recuperación de errores hacia atrás es que se trata de un método generalmente aplicable e independiente de cualquier sistema o proceso específico. En otros términos, puede ser incorporado en (la capa de middleware de) un sistema distribuido como servicio de propósito general.

No obstante, la recuperación de errores hacia atrás también presenta algunos problemas (Singhal y Shivaratri, 1994). En primer lugar, la restauración de un sistema o proceso a un estado previo es por lo general una operación relativamente costosa en cuanto a desempeño. Como se verá en secciones siguientes, a menudo se tiene que realizar mucho trabajo para recuperarse de, por ejemplo, una congelación de proceso o de una falla de sitio. Una forma potencial de salir de este problema es idear mecanismos muy baratos mediante los cuales los componentes simplemente se reinicien. Más adelante regresaremos a este método.

En segundo lugar, como los mecanismos de recuperación de errores son independientes de la aplicación distribuida para la que en realidad se utilizan, no se pueden dar garantías de que una vez ocurrida la recuperación no volverá a suceder la misma falla o una similar. Si tales garantías son requeridas, el manejo de errores a menudo exige que la aplicación entre al bucle de recuperación. En otros términos, la transparencia de fallas totalmente maduras, en general, no puede ser provista por mecanismos de recuperación de errores hacia atrás.

Por último, aunque la recuperación de errores hacia atrás requiere de la marcación de puntos de control, algunos estados simplemente no pueden volver atrás. Por ejemplo, una vez que una persona (posiblemente maliciosa) ha tomado los $1 000 que repentinamente salieron de un cajero automático que estaba funcionando incorrectamente, sólo hay una pequeña posibilidad de que el dinero sea devuelto a la máquina. Asimismo, en la mayoría de los sistemas UNIX, la recuperación a un estado previo después de haber tecleado entusiastamente.

```
rm – fr*
```

pero desde el directorio que funciona erróneamente, puede hacer palidecer a muchas personas. Algunos errores son simplemente irreversibles.

La marcación de puntos de control permite regresar a un estado previo. Sin embargo, a menudo es una operación costosa que puede penalizar severamente el desempeño. En consecuencia, muchos sistemas distribuidos tolerantes a fallas combinan la marcación de puntos de control con el

registro de mensajes. En este caso, después de que se ha tomado un punto de control, un proceso (llamado **registro basado en el remitente**) registra sus mensajes antes de enviarlos. Una solución alternativa es hacer que el proceso receptor registre primero un mensaje entrante antes de entregarlo a la aplicación que esté ejecutando. Este esquema también se conoce como **registro basado en el destinatario**. Cuando un proceso receptor se congela, es necesario restaurarlo al estado más recientemente marcado con un punto de control y, a partir de ahí, *volver a pasar* los mensajes que hayan sido enviados. Por consiguiente, al combinar puntos de control con registro de mensajes es posible recuperar un estado localizado más allá del punto de control más reciente sin el costo de la marcación de puntos de control.

Sigue otra importante distinción entre la marcación de puntos de control y los esquemas que adicionalmente utilizan registros. En un sistema donde sólo se utiliza la marcación de puntos de control, los procesos se restaurarán a un estado marcado con un punto de control. A partir de ahí, su comportamiento puede ser diferente de lo que fue antes de ocurrir la falla. Por ejemplo, como los tiempos de comunicación no son deterministas, ahora pueden ser entregados en un orden diferente, lo que a su vez conduce a que los destinatarios reaccionen de manera distinta. Sin embargo, si el registro de mensajes ocurre, una auténtica recapitulación de los eventos acontecidos desde la última marcación de puntos de control tiene lugar. Tal recapitulación facilita interactuar con el mundo exterior.

Por ejemplo, consideremos el caso en que ocurrió una falla porque un usuario proporcionó una entrada errónea. Si sólo se utiliza la marcación de puntos de control, el sistema tendría que marcar un punto de control antes de aceptar la entrada del usuario para recuperarse a exactamente el mismo estado. Con el registro de mensajes, se puede utilizar una antigua marcación de punto de control, después de lo cual puede ocurrir una recapitulación de eventos hasta el punto en que el usuario deberá proporcionar una entrada. En la práctica, la combinación de marcar algunos puntos de control y el registro de mensajes resulta más eficiente que tener que marcar muchos puntos de control.

Almacenamiento estable

Para recuperarse a un estado previo, es necesario que la información requerida para habilitar la recuperación sea guardada con seguridad. Seguridad en este contexto significa que la recuperación sobreviva a congelaciones de proceso y fallas de sitio, pero quizá también a varias fallas de medios de almacenamiento. El almacenamiento estable desempeña un rol muy importante cuando se trata de recuperación en sistemas distribuidos. Aquí lo analizamos brevemente.

El almacenamiento se presenta en tres categorías. Primero, existe una memoria RAM ordinaria que se borra cuando falla la corriente o una máquina se congela. En segundo lugar tenemos un almacenamiento en disco, el cual sobrevive a fallas de la CPU pero también se puede perder cuando ocurren fallas de cabeza de disco.

Por último, también existe **almacenamiento estable**, el cual está diseñado para sobrevivir a cualquier cosa excepto a calamidades extremas tales como inundaciones o terremotos. El almacenamiento estable puede ser implementado con un par de discos ordinarios, como se muestra en la figura 8-23(a). Cada bloque grabado en el disco 2 es una copia exacta del bloque correspondiente incluido en el disco 1. Cuando un bloque se actualiza, primero se actualiza y verifica el bloque en el disco 1, luego se hace lo mismo en el correspondiente bloque del disco 2.

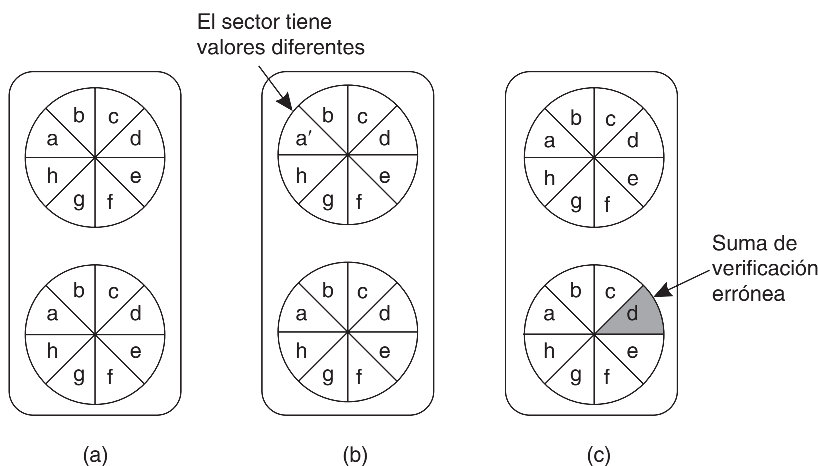


Figura 8-23. (a) Almacenamiento estable. (b) Congelación después de que se actualiza el disco 1. (c) Punto defectuoso.

Supongamos que el sistema se congela después de que el disco 1 se actualiza pero antes de la actualización del disco 2, como ilustra la figura 8-23(b). Al recuperarse, el disco puede ser comparado bloque por bloque. Siempre que dos bloques correspondientes difieren, se puede suponer que el disco 1 es el correcto (porque el disco 1 siempre se actualiza antes que el disco 2), así que el nuevo bloque se copia del disco 1 al disco 2. Cuando se completa el proceso de recuperación, ambos discos nuevamente serán idénticos.

Otro problema potencial es el deterioro espontáneo de un bloque. En un bloque previamente válido, las partículas de polvo o el desgaste general y roturas pueden provocar un error repentino de suma de verificación, sin ninguna causa o advertencia aparente, como se muestra en la figura 8-23(c). Cuando se detecta un error como ese, el bloque defectuoso puede ser regenerado a partir del bloque correspondiente localizado en el otro disco.

A resultas de esta implementación, el almacenamiento estable es muy adecuado para aplicaciones que requieren un alto grado de tolerancia a las fallas, tales como transacciones atómicas. Cuando se escriben datos en almacenamiento estable y luego se leen para comprobar si se escribieron correctamente, la probabilidad de que se pierdan después es extremadamente pequeña.

En las secciones siguientes abordamos con más detalle las marcaciones de puntos de control y el registro de mensajes. Elnozahy y colaboradores (2002) proporcionan un estudio sobre la marcación de puntos de control y el registro en sistemas distribuidos. En Chow y Johnson (1997) se pueden encontrar varios detalles algorítmicos.

8.6.2 Marcación de puntos de control

En un sistema distribuido tolerante a fallas, la recuperación de errores hacia atrás requiere que el sistema guarde con regularidad su estado en almacenamiento estable. En particular, se tiene que registrar un estado global consistente, llamado también **instantánea distribuida**. En una instantánea

distribuida, si un proceso *P* registró la recepción de un mensaje, entonces también deberá haber un proceso *Q* que registró el envío de dicho mensaje. Después de todo, el mensaje debe provenir de alguna parte.

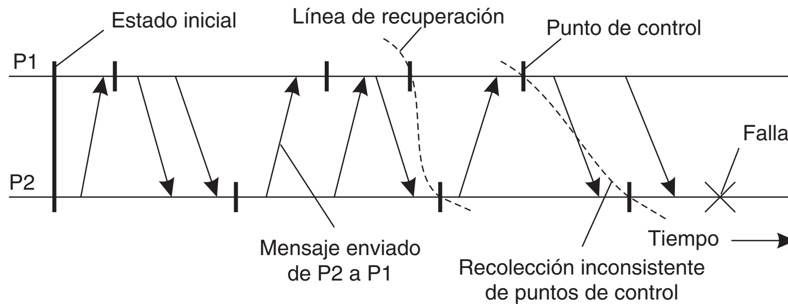


Figura 8-24. Línea de recuperación.

En esquemas de recuperación de errores hacia atrás, de vez en cuando cada proceso guarda su estado en un almacenamiento estable localmente disponible. Para recuperarse después de una falla de proceso o de sistema, se requiere construir un estado global consistente a partir de esos estados locales. En particular, es mejor recuperarse a la instantánea distribuida *más reciente*, también conocida como **línea de recuperación**. En otros términos, una línea de recuperación corresponde a la recolección consistente más reciente de puntos de control, como se muestra en la figura 8-24.

Marcación de puntos de control independiente

Por desgracia, la naturaleza distribuida de la marcación de puntos de control (en la que cada proceso simplemente registra su estado local de vez en cuando de una manera no coordinada) puede hacer difícil encontrar una línea de recuperación. Para descubrir una línea de recuperación se requiere que cada proceso retroceda a su estado más recientemente guardado. Si estos estados locales juntos no conforman una instantánea distribuida, es necesario más retroceso. A continuación, describiremos una forma de encontrar una línea de recuperación. Este proceso de un retroceso en cascada puede conducir a lo que se conoce como **efecto dominó**, y se muestra en la figura 8-25.

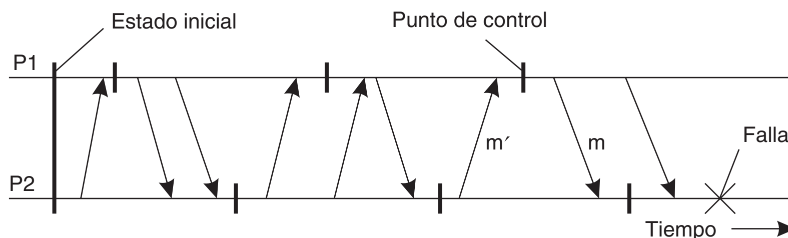


Figura 8-25. Efecto dominó.

Cuando un proceso P_2 se congela, se tiene que recuperar su estado al punto de control más recientemente guardado. En consecuencia, el proceso P_1 también tendrá que ser retrocedido. Por desgracia, los dos estados locales más recientemente guardados no forman un estado global consistente: el estado guardado por P_2 indica la recepción de un mensaje m, pero ningún otro proceso puede ser identificado como su remitente. Por tanto, P_2 tiene que ser retrocedido hasta un estado previo.

Sin embargo, el siguiente estado al cual P_2 es retrocedido tampoco puede ser utilizado como parte de una instantánea distribuida. En este caso, P_1 habrá registrado la recepción del mensaje m', pero no existe ningún evento registrado de que este mensaje esté siendo enviado. Por consiguiente, es necesario retroceder también P_1 a un estado previo. En este ejemplo, resulta que la línea de recuperación es en realidad el estado inicial del sistema.

Como los procesos marcan puntos de control locales independientes entre sí, este método también se conoce como **marcación de puntos de control independiente**. Una solución alternativa es coordinar globalmente la marcación de puntos de control, como veremos a continuación, pero la coordinación requiere sincronización global, la cual puede presentar problemas de desempeño. Otra desventaja de la marcación de puntos de control independiente es que cada almacenamiento local tiene que ser limpiado periódicamente, por ejemplo, haciendo funcionar un recolector de basura distribuido especial. Sin embargo, la desventaja principal radica en el cálculo de la línea de recuperación.

La implementación de marcación de puntos de control independiente requiere que se registren dependencias de tal modo que los procesos puedan retroceder en forma conjunta a un estado global consistente. Con esa finalidad, sea $CP_i(m)$ el punto de control m-ésimo tomado por el proceso P_i. También, sea $INT_i(m)$ el intervalo entre los puntos de control $CP_i(m - 1)$ y $CP_i(m)$.

Cuando el proceso P_i envía un mensaje en el intervalo $INT_i(m)$, incorpora el par (i, m) al proceso receptor. Cuando el proceso P_j recibe un mensaje en el intervalo $INT_j(n)$ junto con el par de índices (i, m), registra entonces la dependencia $INT_i(m) \rightarrow INT_j(n)$. Siempre que P_j toma el punto de control $CP_j(n)$, escribe adicionalmente esta dependencia a su almacenamiento estable local, junto con el resto de la información de recuperación que forma parte de $CP_j(n)$.

Supongamos ahora que en cierto momento se requiere que el proceso P_1 retroceda al punto de control $CP_i(m - 1)$. Para garantizar la consistencia global, debemos asegurarnos de que todos los procesos que hayan recibido mensajes de P_i enviados en el intervalo $INT_i(m)$ sean retrocedidos hasta un estado de punto de control previo a la recepción de los mensajes. En particular, el proceso P_j es nuestro ejemplo, tendrá que ser retrocedido a por lo menos el punto de control $CP_j(n - 1)$. Si CP_j $(n - 1)$ no conduce a un estado globalmente consistente, puede ser que se requiera más retroceso.

El cálculo de la línea de recuperación, requiere un análisis de las dependencias de intervalo registradas por cada proceso cuando se toma un punto de control. Sin entrar en más detalles, resulta que los cálculos son bastante complejos y no justifican la necesidad de marcación de puntos de control independiente en comparación con la marcación de puntos de control coordinada. Además, como resulta ser, a menudo no es la coordinación entre procesos el factor de desempeño dominante, sino los gastos indirectos provenientes de tener que guardar el estado en un almacenamiento local estable. Por consiguiente, la marcación de puntos de control coordinada, más simple que la marcación de puntos de control independiente, a menudo es más popular y presumiblemente permanecerá así incluso cuando los sistemas crezcan a tamaños mucho más grandes (Elnozahy y Planck, 2004).

Marcación de puntos de control coordinada

Como su nombre lo sugiere, en la **marcación de puntos de control coordinada** todos los procesos se sincronizan para escribir conjuntamente su estado en un almacenamiento local estable. La ventaja principal de la marcación de puntos de control coordinada es que el estado guardado es automática y globalmente consistente, de modo que se evite que los retrocesos conduzcan al efecto dominó. El algoritmo de instantánea distribuido que presentamos en el capítulo 6 puede ser utilizado para coordinar la marcación de puntos de control coordinada. Este algoritmo es un ejemplo de coordinación de marcación de puntos de control que no se bloquea.

Una solución más simple es utilizar un protocolo de bloqueo bifásico. Un coordinador primero multitransmite un mensaje *CHECKPOINT_REQUEST* a todos los procesos. Cuando un proceso recibe un mensaje como ese, toma un punto de control local, pone en cola cualquier mensaje subsiguiente entregado a él por la aplicación que se está ejecutando, y confirma al coordinador que ha tomado un punto de control. Cuando el coordinador recibe una confirmación de todos los procesos, multitransmite un mensaje *CHECKPOINT_DONE* para permitir que los procesos (bloqueados) continúen.

Es fácil advertir que este método también conduce a un estado globalmente consistente, porque ningún mensaje entrante es registrado jamás como parte de un punto de control. La razón de esto es que cualquier mensaje que siga a una solicitud de tomar un punto de control no se considera parte del punto de control local. Al mismo tiempo, los mensajes salientes (tal como son entregados al proceso de marcación de puntos de control por la aplicación que se está ejecutando) se ponen localmente en cola hasta que se recibe el mensaje *CHECKPOINT_DONE*.

Una mejora a este algoritmo es la multitransmisión de una petición de punto de control a sólo aquellos procesos que dependen de la recuperación del coordinador, e ignoran a los demás procesos. Un proceso depende del coordinador si recibió un mensaje que está directa o indirectamente relacionado en forma causal con un mensaje que el coordinador había enviado desde el último punto de control. Esto conduce a la noción de **instantánea incremental**.

Para tomar una instantánea incremental, el coordinador multitransmite una petición de marcar un punto de control sólo a aquellos procesos a los que había enviado un mensaje desde la última vez que tomó un punto de control. Cuando un proceso *P* recibe una solicitud como esa, la remite a todos los procesos a los cuales había enviado un mensaje desde el último punto de control, y así sucesivamente. Un proceso remite la solicitud sólo una vez. Cuando todos los procesos han sido identificados, se utiliza una segunda multitransmisión para activar la marcación de puntos de control y permitir que los procesos continúen desde donde se habían quedado.

8.6.3 Registro de mensajes

En consideración a que la marcación de puntos de control es una operación costosa, en especial por lo que se refiere a operaciones implicadas al escribir su estado en un almacenamiento estable, se han buscado técnicas para reducir el número de puntos de control, pero que sigan habilitando la recuperación. En sistemas distribuidos, una técnica importante es el registro de mensajes.

La idea básica que fundamenta el registro de mensajes es que si se puede *repetir* la transmisión de mensajes, aún es posible alcanzar un estado globalmente consistente pero sin tener que

restaurarlo desde un almacenamiento estable. En cambio, se toma un estado con marcación de puntos de control como punto de partida, y todos los mensajes enviados desde entonces simplemente se retransmiten y manejan como corresponde.

Este método funciona bien conforme a la suposición de lo que se llama el **modelo determinístico fragmentado**. En este modelo, se supone que la ejecución de cada proceso se realiza como una serie de intervalos en los que ocurren los eventos. Estos eventos son los mismos que estudiamos en el contexto de la relación de Lamport denominada ocurrencia anterior en el capítulo 6. Por ejemplo, un evento puede ser la ejecución de una instrucción, el envío de un mensaje, y así sucesivamente. En el modelo determinístico fragmentado suponemos que cada intervalo se inicia con un evento no determinístico, tal como la recepción de un mensaje. Sin embargo, desde ese momento en adelante, la ejecución del proceso es completamente determinística. Un intervalo termina con el último evento antes de que ocurra un evento no determinístico.

En efecto, un intervalo puede ser repetido con un resultado conocido, es decir, en una forma completamente determinística, siempre que se repita iniciando con el mismo evento no determinístico como antes. En consecuencia, si se registran todos los eventos no determinísticos presentes en ese modelo, llega a ser posible repetir por completo toda la ejecución de un proceso en una forma determinística.

En consideración a que los registros de mensajes son necesarios para recuperarse de una congelación de proceso de modo que se restaure un estado globalmente consistente, llega a ser importante saber con precisión cuándo tienen que ser registrados los mensajes. Siguiendo el método descrito por Alvisi y Marzullo (1998), resulta que muchos esquemas de registro de mensajes existentes pueden ser caracterizados con facilidad, si nos concentramos en cómo se ocupan de los procesos huérfanos.

Un **proceso huérfano** es un proceso que sobrevive a la congelación de otro proceso, pero cuyo estado es inconsistente con el proceso congelado después de su recuperación. Como un ejemplo, consideremos la situación mostrada en la figura 8-26. El proceso Q recibe los mensajes m_1 y m_2 de los procesos P y R, respectivamente, y luego envía un mensaje m_3 a R. Sin embargo, por contraste con todos los demás mensajes, el mensaje m_2 no es registrado. Si el proceso Q se congela y más tarde se recupera, únicamente los mensajes registrados requeridos para la recuperación de Q se repiten, en nuestro ejemplo, m_1. Como m_2 no fue registrado su transmisión no será repetida, lo cual significa que la transmisión de m_3 tampoco puede tener lugar, figura 8-26.

No obstante, la situación después de la recuperación de Q es inconsistente con aquella que había antes de tal recuperación. En particular, R conserva un mensaje (m_3) que fue enviado antes de su congelación. Tales inconsistencias desde luego que deberán ser evitadas.

Caracterización de los esquemas de registro de mensajes

Para caracterizar los diferentes esquemas de registro de mensajes, se sigue el método descrito en Alvisi y Marzullo (1998). Se considera que cada mensaje m tiene un encabezado que contiene toda la información necesaria para retransmitir m y manejarlo apropiadamente. Por ejemplo, cada encabezado identificará al remitente y al destinatario, pero también un número en secuencia para

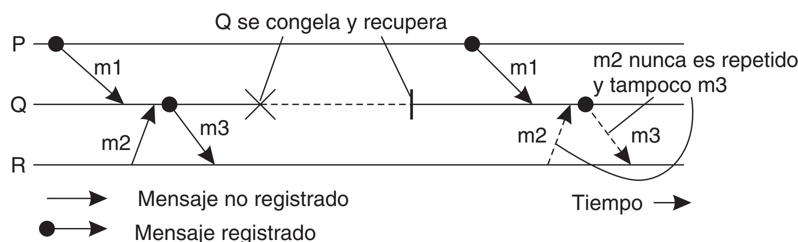


Figura 8-26. Repaso incorrecto de los mensajes después de la recuperación, conduce a un proceso huérfano.

reconocerlo como duplicado. Además, se puede agregar un número de entrega para decidir con exactitud cuándo deberá ser entregado a la aplicación receptora.

Se dice que un mensaje es **estable** si ya no se puede perder, por ejemplo, porque fue escrito en un almacenamiento estable. Por tanto, los mensajes estables pueden ser utilizados para recuperación repitiendo su transmisión.

Cada mensaje *m* conduce a un conjunto *DEP*(*m*) de procesos que dependen de la entrega de *m*. En particular, *DEP*(*m*) se compone de aquellos procesos a los cuales *m* ha sido entregado. Además, si otro mensaje *m′* es causalmente dependiente de la entrega de *m*, y *m′* ha sido entregado a un proceso *Q*, entonces *Q* también estará contenido en *DEP*(*m*). Observemos que *m′* es causalmente dependiente de la entrega de *m*, si fuera enviado por el mismo proceso que previamente entregó *m*, o del que había entregado otro mensaje que era causalmente dependiente de la entrega de *m*.

El conjunto *COPY*(*m*) se compone de aquellos procesos que tienen una copia de *m*, pero que (aún) no están en su almacenamiento local estable. Cuando un proceso *Q* entrega el mensaje *m*, también se vuelve miembro de *COPY*(*m*). Observe que *COPY*(*m*) se compone de aquellos procesos que podrían entregar una copia de *m* que pueda ser utilizada para retransmitir *m*. Si todos estos procesos se congelan, la retransmisión de *m* desde luego que no es factible.

Al utilizar estas notaciones, ya es fácil definir con precisión qué es un proceso huérfano. Supongamos que en un sistema distribuido algunos procesos se acaban de congelar. Sea *Q* uno de los procesos sobrevivientes. El proceso *Q* es un proceso huérfano si existe un mensaje *m*, de tal suerte que *Q* esté contenido en *DEP*(*m*), mientras que al mismo tiempo cada proceso incluido en *COPY*(*m*) se ha congelado. En otros términos, aparece un proceso huérfano que depende de *m*, pero no hay forma de repetir la transmisión de *m*.

Para evitar procesos huérfanos, se tiene que garantizar que si cada proceso incluido en *COPY*(*m*) se congela, entonces no queda ningún proceso sobreviviente en *DEP*(*m*). Es decir, todos los procesos incluidos en *DEP*(*m*) también deberán haberse congelado. Es posible hacer que se cumpla esta condición si se puede garantizar que siempre que un proceso se vuelva miembro de *DEP*(*m*), también se vuelva miembro de *COPY*(*m*). Esto es, siempre que un proceso se vuelva dependiente de la entrega de *m*, conservará una copia de *m*.

Existen esencialmente dos métodos que se pueden seguir. El primero, está representado por los que se llaman **protocolos de registro pesimistas**. Estos protocolos tienen el cuidado de que por

cada mensaje *no estable m*, existe cuando mucho un proceso dependiente de *m*. En otros términos, los protocolos de registro pesimistas garantizan que cada mensaje *no estable m* sea entregado a cuando mucho un proceso. Observemos que en cuanto *m* es entregado a, por ejemplo el proceso *P*, *P* se vuelve miembro de *COPY*(*m*).

Lo peor que puede suceder es que el proceso *P* se congele sin que *m* haya sido registrado. Con registro pesimista, no se permite que *P* envíe ningún mensaje después de la entrega de *m* sin primero haberse asegurado de que *m* se escribió en un almacenamiento estable. Por consiguiente, ningunos otros procesos se volverán dependientes de la entrega de *m* a *P* sin que haya la posibilidad de retransmitir *m*. De esta manera, siempre se evita un proceso huérfano.

Por contraste, en un **protocolo de registro optimista**, el trabajo propiamente dicho se realiza *después* de que ocurre una congelación. En particular, supongamos que para algún mensaje *m* cada proceso incluido en *COPY*(*m*) se ha congelado. En un método optimista, cualquier proceso huérfano en *DEP*(*m*) es retrocedido a un estado en el cual ya no pertenece a *DEP*(*m*). Desde luego, los protocolos de registro optimistas no deben perder de vista las dependencias, lo cual complica su implementación.

Como se señala en Elnozahy y colaboradores (2002), el registro pesimista resulta mucho más simple que los métodos optimistas, por ello es la forma preferida de registro de mensajes en el diseño de sistemas distribuidos prácticos.

8.6.4 Computación orientada a la recuperación

Una forma relacionada de manejar la recuperación es, en esencia, empezar de nuevo. El principio fundamental hacia esta forma de ocultar las fallas es que puede resultar mucho más barato optimizar para recuperación, esto es, buscar sistemas que no fallen durante mucho tiempo. Este método también se conoce como **computación orientada a la recuperación** (Candea y cols., 2004a).

Existen diferentes sabores de la computación orientada a la recuperación. Uno es simplemente reiniciar (una parte de un sistema), y ha sido explorado para reiniciar servidores de internet (Candea y cols., 2004b, 2006). Para poder reiniciar sólo una parte del sistema, es crucial localizar apropiadamente la falla. En ese punto, reiniciar significa simplemente borrar todas las instancias de los componentes identificados, junto con los hilos que operan en ellos y (a menudo) simplemente reiniciar las solicitudes asociadas. Observemos que la propia localización de fallas puede ser un ejercicio no trivial (Steinder y Sethi, 2004).

Para habilitar el reinicio como técnica de recuperación práctica se requiere que los componentes estén grandemente desacoplados en el sentido de que existan pocas o ningunas dependencias entre los distintos componentes. Si existen fuertes dependencias, entonces localizar y analizar la falla puede requerir el reinicio de todo el servidor hasta el punto en el cual la aplicación de técnicas de recuperación tradicionales como las estudiadas aquí puedan ser más eficientes.

Otro sabor de la computación orientada a la recuperación es aplicar la marcación de puntos de control y técnicas de recuperación, pero continuar la ejecución en un ambiente cambiado. La idea básica en este caso es que muchas fallas simplemente pueden evitarse si a los programas se les permite más espacio de bufer, poner en cero la memoria antes de asignarla, cambiando el orden de la

entrega de los mensajes (en tanto no afecte la semántica), etc. (Qin y cols., 2005). La idea fundamental es solucionar fallas de software (en tanto que muchas de las técnicas examinadas hasta ahora están encaminadas a resolver, o están basadas en fallas de hardware). Como la ejecución de un software es altamente determinística, cambiar el ambiente de ejecución puede salvar el día, pero, naturalmente, sin reparar nada.

8.7 RESUMEN

La tolerancia a fallas es un tema importante en el diseño de sistemas distribuidos. La tolerancia a fallas se define como la característica mediante la cual un sistema puede ocultar la ocurrencia y la reparación de fallas. En otros términos, un sistema es tolerante a fallas si puede continuar operando en presencia de éstas.

Existen varios tipos de fallas. Una falla de congelación ocurre cuando un proceso simplemente se detiene. Una falla por omisión ocurre cuando un proceso no responde a las solicitudes entrantes. Cuando un proceso responde demasiado rápido o demasiado tarde a una solicitud, se dice que exhibe una falla de temporización. Responder a una solicitud entrante, pero de manera equivocada, es un ejemplo de falla de respuesta. Las fallas más difíciles de manejar, llamadas fallas arbitrarias o bizantinas, son aquellas mediante las cuales un proceso exhibe cualquier clase de falla.

La redundancia es la técnica fundamental requerida para lograr la tolerancia a fallas. Cuando se aplica a procesos, la noción de grupos de procesos se vuelve importante. Un grupo de procesos se compone de varios procesos que cooperan estrechamente para proporcionar un servicio. En grupos de procesos tolerantes a fallas, uno o más procesos pueden fallar sin afectar la disponibilidad del servicio que ofrece el grupo. A menudo, es necesario que la comunicación dentro del grupo sea altamente confiable y que se adhiera a propiedades de atomicidad y ordenamiento estrictas para lograr la tolerancia a fallas.

La comunicación de grupo confiable, también llamada multitransmisión confiable, se presenta en formas diferentes. En tanto los grupos sean relativamente pequeños, la implementación de la confiabilidad es factible. Sin embargo, cuando grupos muy grandes tienen que ser soportados, la escalabilidad de la multitransmisión confiable se vuelve problemática. El tema clave al lograr la escalabilidad es reducir el número de mensajes de retroalimentación mediante los cuales los receptores informan sobre la recepción (no) exitosa de un mensaje multitransmitido.

Las cosas empeoran cuando se tiene que proporcionar atomicidad. En protocolos de multitransmisión atómica, es esencial que cada miembro del grupo tenga la misma visión con respecto a qué miembros fue entregado un mensaje multitransmitido. La multitransmisión atómica puede ser formulada con precisión en función de un modelo de ejecución síncrono virtual. En esencia, este modelo presenta límites entre cuáles miembros del grupo no cambian y qué mensajes son transmitidos confiablemente. Un mensaje nunca puede cruzar un límite.

Los cambios de membresía al grupo son un ejemplo en el que cada proceso tiene que estar de acuerdo con la misma lista de miembros. Semejante acuerdo puede alcanzarse por medio de protocolos de realización, de los cuales el de dos fases es el más aplicado. En un protocolo de realización bifásico, un coordinador primero investiga si todos los procesos están de acuerdo en realizar

la misma operación (es decir, si todos están de acuerdo en realizarla), y en una segunda ronda multitransmite el resultado de dicha encuesta. Se utiliza un protocolo de realización trifásico para ocuparse de la congelación del coordinador sin tener que bloquear todos los procesos para llegar a un acuerdo hasta que el coordinador se recupere.

En sistemas tolerantes a fallas, la recuperación se logra invariablemente marcando con puntos de control el estado del sistema en forma regular. La marcación de puntos de control es completamente distribuida. Desafortunadamente, la toma de un punto de control es una operación cara. Para mejorar el desempeño, muchos sistemas distribuidos combinan la marcación de puntos de control con el registro de mensajes. Registrando la comunicación entre los procesos, llega a ser posible repetir la ejecución del sistema después de ocurrida una congelación.

PROBLEMAS

1. A menudo se requiere que los sistemas confiables proporcionen un alto grado de seguridad. ¿Por qué?

2. ¿Qué hace que el modelo de falla por detención, en el caso de fallas por congelación, sea tan difícil de implementar?

3. Considere un navegador web que regresa una anticuada página guardada en caché en lugar de una más reciente que había sido actualizada en el servidor. ¿Es ésta una falla?, de ser así, ¿qué clase de falla?

4. ¿Puede el modelo de redundancia modular triple descrito en el texto manejar fallas bizantinas?

5. ¿Cuántos elementos con falla (dispositivos más votantes) puede manejar la figura 8-2? Proporcione un ejemplo del peor caso que puede ser disfrazado.

6. ¿Generaliza la TMR a cinco elementos por grupo en lugar de tres? De ser así, ¿qué propiedades tiene?

7. Para cada una de las siguientes aplicaciones, ¿considera usted que es mejor la semántica de por lo menos una vez o la semántica de a lo más una vez? Analícelo.

(a) Leer y escribir archivos desde un servidor de archivos.

(b) Compilar un programa.

(c) Realizar operaciones bancarias a distancia.

8. Con RPC asíncronas, un cliente se bloquea hasta que su solicitud ha sido *aceptada* por el servidor. ¿Hasta qué grado afectan las fallas a la semántica de RPC asíncronas?

9. Proporcione un ejemplo en el cual la comunicación de grupo no requiera en absoluto de un ordenamiento de los mensajes.

10. En multitransmisión confiable, ¿siempre es necesario que la capa de comunicación conserve una copia de un mensaje para propósitos de retransmisión?

11. ¿A qué grado es importante la escalabilidad de la multitransmisión atómica?

12. En el texto sugerimos que la multitransmisión atómica puede salvarle el día cuando se trata de realizar actualizaciones de un conjunto de procesos acordado. ¿A qué grado se puede garantizar que cada actualización efectivamente sea realizada?

13. La sincronía virtual es análoga a la consistencia débil en almacenes de datos distribuidos, con cambios de vista de grupo que actúan como puntos de sincronización. En este contexto, ¿cuál sería el análogo de la consistencia fuerte?

14. ¿Cuáles son los ordenamientos de entrega permisibles para la combinación de orden FIFO y multitransmisión en orden total en la figura 8-15?

15. Adapte el protocolo para instalar la vista siguiente G_i+1 en el caso de sincronía virtual de modo que pueda tolerar fallas de proceso.

16. En el protocolo de realización bifásico, ¿por qué el bloqueo nunca puede ser eliminado por completo, incluso cuando los participantes eligen un nuevo coordinador?

17. En nuestra explicación sobre realización trifásica, parece que la realización de una transacción está basada en votación mayoritaria. ¿Es esto cierto?

18. En un modelo de ejecución determinístico fragmentado, ¿es suficiente con registrar sólo mensajes o también se tienen que registrar otros eventos?

19. Explique cómo se puede utilizar el registro de escritura adelantada en transacciones distribuidas para recuperarse de fallas.

20. ¿Tiene que tomar puntos de control un servidor sin estado?

21. El registro de mensajes basado en el receptor generalmente se considera mejor que el registro basado en el remitente. ¿Por qué?

9

SEGURIDAD

El último principio de sistemas distribuidos que se analiza es el de seguridad. La seguridad de ningún modo es el punto menos importante. Sin embargo, se podría argumentar que es uno de los más difíciles, ya que la seguridad tiene que estar presente en todo el sistema. Una sola falla de diseño con respecto a la seguridad puede hacer que todas las medidas de seguridad sean inútiles. En este capítulo, prestamos atención especial a los diversos mecanismos incorporados en general en los sistemas distribuidos para dar soporte a la seguridad.

Primero presentamos los temas básicos de seguridad. La incorporación de diversos mecanismos de seguridad en un sistema en realidad no tiene sentido a menos que se sepa cómo han de ser utilizados y contra qué. Esto implica el conocimiento de la política de seguridad a aplicar. La noción de una política de seguridad, junto con algunos temas de diseño de los mecanismos que ayuden a aplicarla, se aborda primero. También se toca brevemente la criptografía necesaria.

La seguridad en los sistemas distribuidos se divide a menudo en dos partes. Una parte tiene que ver con la comunicación entre usuarios y procesos, los cuales posiblemente residen en máquinas diferentes. El mecanismo principal para garantizar la comunicación segura es un canal seguro. Los canales seguros, y más específicamente, la integridad y la confidencialidad del mensaje de autenticación, se abordan en otra sección.

La otra parte del tema de seguridad se ocupa de la autorización, la cual garantiza que un proceso obtenga sólo aquellos derechos de acceso a los recursos de un sistema distribuido para los que tiene autorización. La autorización se trata en una sección aparte que aborda el control de acceso. Además de los mecanismos de acceso tradicionales, también se presta atención al control de acceso que tiene que ver con códigos móviles tales como agentes.

Los canales seguros y el control de acceso requieren mecanismos de distribución de claves criptográficas, pero también mecanismos para agregar o eliminar usuarios de un sistema. Estos temas son abordados por lo que se conoce como gestión de seguridad. En una sección aparte, analizamos los temas relacionados con claves criptográficas, gestión de grupo seguro, y otorgamiento de certificados que comprueben que el propietario está autorizado para acceder a recursos específicos.

9.1 INTRODUCCIÓN A LA SEGURIDAD

Iniciamos la descripción de la seguridad en sistemas distribuidos con el repaso de algunos temas generales de seguridad. En primer lugar, es necesario definir qué es un sistema seguro. Se distinguen las *políticas* de seguridad de los *mecanismos* de seguridad, y se estudia un sistema de área amplia Globus para el cual ha sido formulada explícitamente una política de seguridad. En segundo lugar, se consideran algunos temas de diseño generales sobre sistemas seguros. Por último, se analizan algunos algoritmos criptográficos, los cuales desempeñan un rol primordial en el diseño de protocolos de seguridad.

9.1.1 Amenazas, políticas y mecanismos de seguridad

En un sistema de computadora, la seguridad está fuertemente relacionada con la noción de confiabilidad. Informalmente, un sistema de computadora confiable es uno en el que justificadamente confiamos nos suministrará sus servicios (Laprie, 1995). Como se mencionó en el capítulo 7, la confiabilidad incluye disponibilidad, consistencia, seguridad, y sustentabilidad. Sin embargo, si se va a confiar en un sistema de computadora, entonces la confidencialidad y la integridad también deberán ser tomadas en cuenta. **Confidencialidad** se refiere a la propiedad de un sistema de computadora mediante la cual su información se pone sólo al alcance de personas autorizadas. La **integridad** es la característica de que las modificaciones de los activos de un sistema sólo pueden hacerse en una forma autorizada. Es decir, en un sistema de computadora seguro, las modificaciones inapropiadas deben ser detectables y recuperables. Los activos principales de cualquier sistema de computadora se encuentran en su equipo físico, sus programas, y sus datos.

Otra forma de considerar la seguridad en los sistemas de computadora es como un intento de proteger los servicios y datos que ofrece contra **amenazas de seguridad**. Existen cuatro tipos de amenazas de seguridad a considerar (Pfleeger, 2003):

1. Intercepción
2. Interrupción
3. Modificación
4. Fabricación

El concepto de intercepción se refiere a la situación en la que una persona no autorizada ha logrado acceder a los servicios o datos. Un ejemplo típico de intercepción es aquel donde la comunicación

entre dos personas es escuchada por alguien más. También ocurre intercepción cuando los datos son ilegalmente copiados, por ejemplo, después de violar el directorio privado de una persona contenido en un sistema de archivo.

Un ejemplo de interrupción es cuando un archivo se corrompe o pierde. De modo más general, la interrupción se refiere a la situación en la que los servicios o datos ya no están disponibles, se inutilizan o destruyen y así sucesivamente. En este caso, la neutralización de ataques mediante los cuales alguien maliciosamente intenta hacer que otras personas no tengan acceso a un servicio es una amenaza de seguridad clasificada como interrupción.

Las modificaciones implican el cambio no autorizado o la manipulación de un servicio de tal forma que ya no se adhiera a sus especificaciones originales. Ejemplos de modificaciones incluyen la intercepción y el cambio subsecuente de los datos transmitidos, la manipulación de entradas de bases de datos, y el cambio de un programa de modo que registre en secreto las actividades de su usuario.

Fabricación se refiere a la situación en que se generan datos o actividades adicionales que normalmente no existirían. Por ejemplo, un intruso podría intentar agregar una entrada a un archivo de contraseña o base de datos. Asimismo, en ocasiones es posible violar un sistema reponiendo mensajes previamente enviados. Más adelante en este capítulo se verán tales ejemplos.

Es importante advertir que la interrupción, la modificación, y la fabricación pueden ser consideradas como una forma de falsificación de datos.

La simple declaración de que un sistema deberá ser capaz de autoprotegerse contra posibles amenazas de seguridad no es forma de construir en realidad un sistema seguro. Lo primero que hay que hacer es describir los requerimientos de seguridad, es decir, establecer una política de seguridad. Una **política de seguridad** describe con precisión qué acciones le están permitidas a las entidades de un sistema y cuáles están prohibidas. Las entidades incluyen usuarios, servicios, datos, máquinas, etc. Un vez delineada una política de seguridad, es posible concentrarse en el **mecanismo de seguridad** mediante el cual pueda ser aplicada. Mecanismos de seguridad importantes son:

1. Cifrado
2. Autenticación
3. Autorización
4. Auditoría

El cifrado es fundamental para la seguridad de una computadora. El cifrado transforma los datos en algo que un atacante no puede entender. En otras palabras, el cifrado permite implementar la confidencialidad de los datos. Además, permite verificar si los datos han sido modificados. También apoya las verificaciones de integridad.

La autenticación se utiliza para verificar la identidad pretendida de un usuario, cliente, servidor, anfitrión u otra entidad. En el caso de clientes, la premisa básica es que antes de que un servidor comience a realizar cualquier trabajo a favor de un cliente, el servicio debe aprenderse la identidad

del cliente (a menos que el servicio esté disponible para todos). Típicamente, los usuarios son autentificados por medio de contraseñas, aunque existen muchas otras formas de hacerlo.

Después de que un cliente ha sido autentificado, es necesario verificar si está autorizado para realizar la acción solicitada. Un ejemplo típico es el acceso a los registros de una base de datos médica. Según quien sea el que accede a la base de datos, se puede permitir que lea los registros, que modifique ciertos campos de un registro, o que agregue o elimine un registro.

Se utilizan herramientas de auditoría para rastrear a qué tuvieron acceso los clientes y de qué forma lo hicieron. Aun cuando la auditoría no protege realmente contra amenazas de seguridad, sus registros pueden resultar extremadamente útiles para analizar una infracción de seguridad y tomar medidas subsecuentes contra intrusos. Por eso, en general, los intrusos se cuidan de no dejar evidencias que con el tiempo pudieran dejar al descubierto su identidad. En este sentido, el registro de acceso hace que un ataque se convierta en un negocio riesgoso.

Ejemplo: la arquitectura de seguridad Globus

La noción de política de seguridad y el rol que los mecanismos de seguridad desempeñan en los sistemas distribuidos para aplicar tales políticas a menudo se entiende mejor analizando un ejemplo concreto. Consideremos la política de seguridad definida para el sistema de área amplia Globus (Chervenak y cols., 2000). Globus es un sistema que soporta cálculos distribuidos a gran escala en los cuales se utilizan al mismo tiempo muchos anfitriones, archivos, y otros recursos para realizar un cálculo. Tales ambientes también se conocen como mallas computacionales (Foster y Kesselman, 2003). En estas mallas los recursos a menudo están localizados en dominios administrativos diferentes que pueden ubicarse en distintas partes del mundo.

Como usuarios y recursos son muchos y se encuentran ampliamente esparcidos a través de diferentes dominios administrativos, la seguridad resulta esencial. Para idear y utilizar apropiadamente los mecanismos de seguridad es necesario entender exactamente qué tiene que estar protegido y cuáles son las suposiciones con respecto a la seguridad. Para simplificar las cosas, la política de seguridad para Globus impone los siguientes ocho enunciados, los cuales se explican a continuación (Foster y cols., 1998):

1. El ambiente se compone de múltiples dominios administrativos.
2. Las operaciones locales (es decir, operaciones que se realizan sólo dentro de un solo dominio) están sujetas a una política de seguridad de dominio local.
3. Las operaciones globales (es decir, operaciones que implican varios dominios) requieren que el iniciador sea conocido en cada uno de los dominios donde la operación se realiza.
4. Las operaciones entre entidades de diferentes dominios requieren autenticación mutua.
5. La autenticación global reemplaza a la local.

6. El control de acceso a los recursos está sujeto a la seguridad local solamente.

7. Los usuarios pueden delegar derechos a procesos.

8. Un grupo de procesos localizados en el mismo dominio puede compartir credenciales.

Globus supone que el ambiente se compone de múltiples dominios administrativos, donde cada dominio dispone de su propia política de seguridad local. Se presume que las políticas locales no pueden ser cambiadas sólo porque el dominio participa en Globus, ni la política global de Globus puede anular las decisiones de seguridad local. Por consiguiente, en Globus la seguridad se autolimitará a operaciones que afectan múltiples dominios.

Algo relacionado con este tema es que Globus asume que las operaciones consideradas totalmente locales en relación con un dominio están sujetas sólo a la política de seguridad de éste. En otras palabras, si una operación es iniciada y realizada dentro de un solo dominio, todas las cuestiones de seguridad se realizarán solamente por medio de medidas de seguridad locales. Globus no impondrá medidas adicionales.

La política de seguridad de Globus estipula que las solicitudes de operación pueden ser iniciadas global o localmente. El iniciador, ya sea un usuario o un proceso que actúa a nombre de un usuario, debe ser localmente conocido dentro de cada dominio donde dicha operación se realiza. Por ejemplo, un usuario puede tener un nombre global correlacionado con nombres locales en un dominio específico. Cómo ocurra exactamente la correlación se deja a cada dominio.

Una importante estipulación de política es que las operaciones entre entidades ubicadas en diferentes dominios requieren autenticación mutua. Esto significa, por ejemplo, que si el usuario de un dominio utiliza un servicio de otro dominio, entonces su identidad tendrá que ser verificada. Igualmente importante es que el usuario esté seguro de que está utilizando el servicio que supone estar utilizando. Se volverá a la autenticación, con mayor amplitud, más adelante en este capítulo.

Los dos temas de seguridad precedentes se combinan en el siguiente requerimiento de seguridad. Si la identidad de un usuario ha sido verificada, y si el usuario también es conocido localmente en un dominio, entonces puede actuar como autenticado para dicho dominio local. Esto significa que Globus requiere que sus medidas de autenticación dentro de todo el sistema sean suficientes para considerar que un usuario ya ha sido autenticado para un dominio remoto (donde dicho usuario es conocido) cuando acceda a los recursos de dicho dominio. No se requerirá autenticación adicional por parte del dominio local.

Una vez que el usuario (o el proceso que actúa a nombre de un usuario) ha sido autenticado, sigue siendo necesario verificar los derechos de acceso exactos con respecto a los recursos. Por ejemplo, un usuario que desee modificar un archivo primero tiene que ser autenticado, tras de lo cual se puede comprobar si en realidad tiene permiso o no de modificar el archivo. La política de seguridad de Globus establece que tales decisiones de control de acceso son por completo locales dentro del dominio donde se localiza el recurso al que se tuvo acceso.

Para explicar el séptimo enunciado, consideremos un agente móvil de Globus que realiza una tarea al iniciar varias operaciones en diferentes dominios, una tras otra. Tal agente puede requerir mucho tiempo para completar su tarea. Para no tener que comunicarse con el usuario a cuyo nombre

el agente está actuando, Globus requiere que los procesos puedan ser delegados a un subconjunto de derechos de usuario. En consecuencia, con la autenticación de un agente y la subsiguiente comprobación de sus derechos, Globus deberá ser capaz de permitir que un agente inicie una operación sin tener que ponerse en contacto con su propietario.

Como estipulación final de su política, Globus requiere que los grupos de procesos que se ejecutan con un solo dominio y actúan a nombre del mismo usuario puedan compartir un solo conjunto de credenciales. Como se explicará enseguida, la autenticación requiere de credenciales. Esta estipulación abre la puerta a soluciones escalables de autenticación puesto que no demanda que cada proceso realice su propio conjunto único de credenciales.

La política de seguridad de Globus permite que sus diseñadores se concentren en el desarrollo de una solución integral de seguridad. Suponiendo que cada dominio aplica su propia política de seguridad, Globus se concentra sólo en amenazas de seguridad que implican múltiples dominios. En particular, la política de seguridad indica que los temas de diseño importantes son la representación de un usuario en un dominio remoto y la asignación de recursos de un dominio remoto a un usuario o a su representante. En consecuencia, lo que Globus requiere principalmente son los mecanismos de autenticación entre dominios y hacer que un usuario sea reconocido en dominios remotos.

Para este propósito, se introducen dos tipos de representantes. Un **proxy de usuario** es un proceso que tiene permiso de actuar a nombre de un usuario durante un lapso de tiempo limitado. Los recursos están representados por proxys de recurso. Un **proxy de recurso** es un proceso que se ejecuta dentro de un dominio específico utilizado para transformar las operaciones globales sobre un recurso en operaciones locales que cumplan con la política de seguridad del dominio en particular. Por ejemplo, un proxy de usuario se comunica normalmente con un proxy de recurso cuando se requiere acceso a sus recursos.

La arquitectura de seguridad Globus, en esencia, se compone de entidades tales como usuarios, proxys de usuario, proxys de recurso y procesos generales. Estas entidades se encuentran en dominios e interactúan entre sí. En particular, la arquitectura de seguridad define cuatro protocolos diferentes, como se ilustra en la figura 9-1 [vea también Foster y cols. (1998)].

El primer protocolo describe con precisión cómo un usuario puede crear un proxy y delegarle derechos. En particular, para permitir que el proxy de usuario actúe a nombre de su usuario, el usuario otorga al proxy un conjunto apropiado de credenciales.

El segundo protocolo especifica cómo un proxy de usuario puede solicitar la asignación de un recurso ubicado en un dominio remoto. En esencia, el protocolo le indica a un proxy de recurso crear un proceso en el dominio remoto una vez que la autenticación mutua haya ocurrido. Dicho proceso representa al usuario (exactamente como lo hizo el proxy de usuario), aunque opere en el mismo dominio que el recurso solicitado. Se permite que el proceso acceda al recurso de acuerdo con las decisiones de control de acceso locales referentes a dicho dominio.

Un proceso creado en un dominio remoto puede iniciar cálculos adicionales en otros dominios. Por consiguiente, se requiere de un protocolo para asignar recursos en un dominio remoto conforme a lo solicitado por un proceso diferente de un proxy de usuario. En el sistema Globus, este tipo de asignación se realiza por intermediación del proxy de usuario, al permitir que un proceso haga que su proxy de usuario asociado solicite la asignación de recursos, siguiendo en esencia el segundo protocolo.

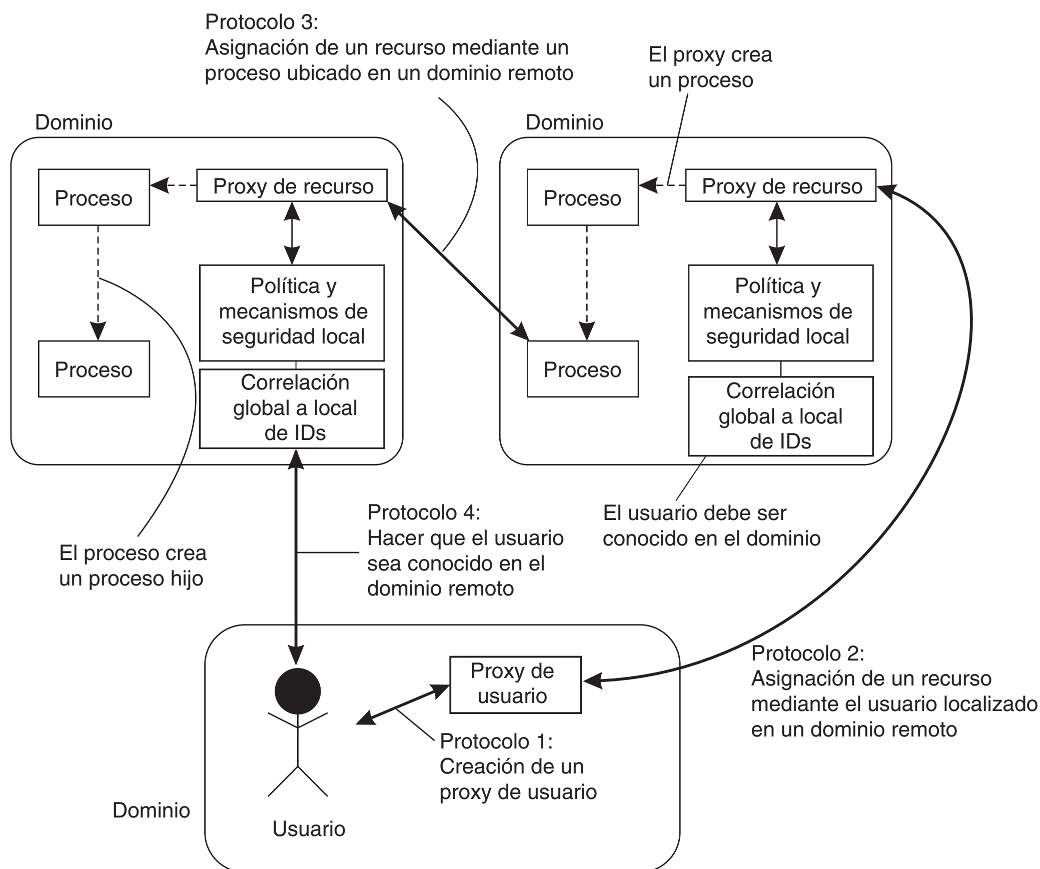


Figura 9-1. Arquitectura de seguridad Globus.

El cuarto y último protocolo de la arquitectura de seguridad Globus es la forma en que un usuario puede hacerse conocido en un dominio. Suponiendo que un usuario tiene una cuenta en un dominio, lo que debe establecerse es que las credenciales implementadas en todo el sistema, tal como aparecen en el proxy de usuario, sean convertidas automáticamente en credenciales reconocidas por el dominio específico. El protocolo decreta cómo puede ser registrada por el usuario la correlación entre las credenciales globales y locales en una mesa de correlación local con respecto al dominio específico.

En Foster y colaboradores (1998) se describen detalles específicos de cada protocolo. La cuestión importante en este caso, es que la arquitectura de seguridad Globus refleje su política de seguridad como se estableció previamente. Los mecanismos utilizados para poner en ejecución esta arquitectura, en particular los protocolos mencionados, son comunes a muchos sistemas distribuidos y se analizan extensamente en este capítulo. La dificultad principal al diseñar sistemas distribuidos seguros no radica en los mecanismos de seguridad, sino en la decisión de cómo tienen que

ser utilizados para aplicar una política de seguridad. En la siguiente sección, analizamos algunas de estas decisiones de diseño.

9.1.2 Temas de diseño

Un sistema distribuido, o cualquier sistema de computadora, debe proporcionar servicios de seguridad mediante los cuales pueda ser implementada una amplia gama de políticas de seguridad. Existen varios temas de diseño importantes que deben ser tomados en cuenta cuando se implementen servicios de seguridad para propósitos generales. En las páginas siguientes analizamos tres de estos temas: enfoque del control, organización en capas de los mecanismos de seguridad, y simplicidad [vea también Gollmann (2006)].

Enfoque del control

Cuando se considera la protección de una aplicación (posiblemente distribuida), existen básicamente tres métodos diferentes que pueden ser seguidos, como se muestra en la figura 9-2. El primer método consiste en concentrarse directamente en la protección de los datos asociados con la aplicación. Directamente significa que independientemente de las diversas operaciones que posiblemente pueden ser realizadas en un rubro de datos, el interés principal es garantizar su integridad. En general, este tipo de protección ocurre en sistemas de bases de datos en los cuales se pueden formular varias restricciones de integridad que sean automáticamente verificadas cada vez que se modifique un rubro de datos [vea, por ejemplo, Doorn y Rivero (2002)].

El segundo método es concentrarse en la protección al especificar con exactitud qué operaciones pueden ser invocadas, y por quién, cuando tenga que accederse a ciertos datos o recursos. En este caso, el enfoque del control está fuertemente relacionado con los mecanismos de control de acceso, los cuales se analizan extensamente más adelante en este capítulo. Por ejemplo, en un sistema basado en objetos, se puede decidir especificar —en cuanto a cada método— qué se pone a la disposición de los clientes y a qué clientes se les permite invocarlo. De modo alterno, se pueden aplicar métodos de control de acceso a toda la interfaz ofrecida por un objeto, o a todo el objeto. Este método permite, por tanto, varias granularidades del control de acceso.

Un tercer método es enfocarse directamente en los usuarios tomando medidas por las cuales sólo personas específicas tienen acceso a la aplicación, independientemente de las operaciones que deseen realizar. Por ejemplo, en un banco, una base de datos puede estar protegida negando el acceso a cualquier persona excepto a la alta gerencia y a las personas específicamente autorizadas para acceder a ella. Como otro ejemplo, en muchas universidades, ciertos datos y aplicaciones sólo pueden ser utilizados por los profesores y miembros del personal, en tanto que no se permite el acceso a los estudiantes. En realidad, el control está enfocado en definir los **roles** que los usuarios tienen, y una vez que el rol de un usuario ha sido verificado se le permite o niega el acceso a un recurso. Como parte del diseño de un sistema seguro es necesario, por tanto, definir los roles que las personas pueden tener y proporcionar mecanismos para dar soporte al control de acceso basado en roles. Más adelante en este capítulo regresaremos al tema de los roles.

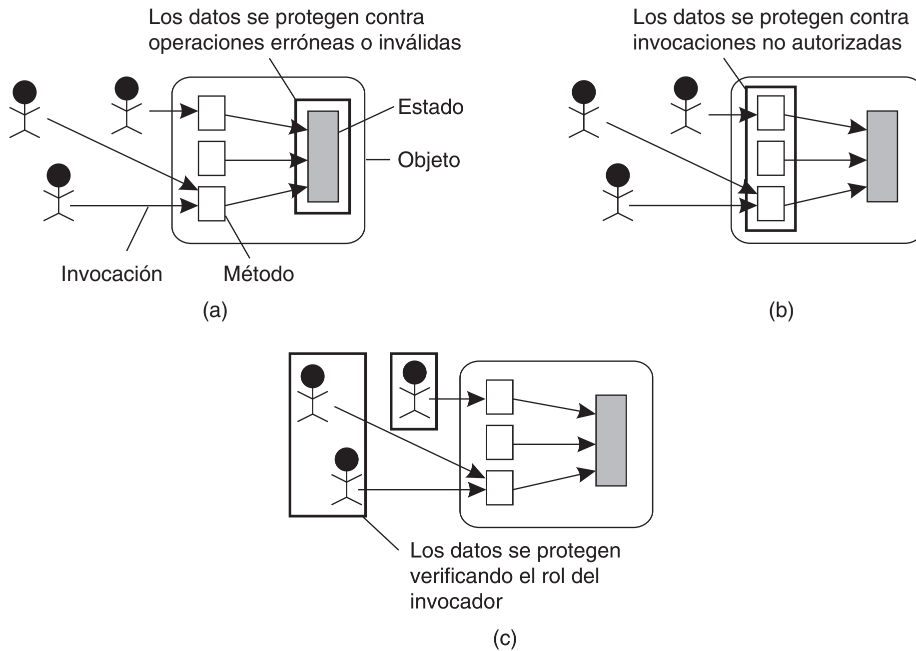


Figura 9-2. Tres métodos de protección contra amenazas de seguridad: (a) Protección contra operaciones inválidas. (b) Protección contra invocaciones no autorizadas. (c) Protección contra usuarios no autorizados.

Organización en capas de los mecanismos de seguridad

Un tema importante al diseñar sistemas seguros es decidir en qué nivel deben ser colocados los mecanismos de seguridad. En este contexto, un nivel está relacionado con la organización lógica de un sistema en varias capas. Por ejemplo, las redes de computadoras a menudo se organizan en capas siguiendo un modelo de referencia, como se vio en el capítulo 4. En el capítulo 1 se presentó la organización de sistemas distribuidos compuestos por capas distintas para aplicaciones, middleware, servicios del sistema operativo y el kernel (núcleo) del sistema operativo. La combinación de la organización en capas de redes de computadoras y sistemas distribuidos conduce aproximadamente a lo mostrado en la figura 9-3.

En esencia, la figura 9-3 separa los servicios de propósito general de los servicios de comunicación. Esta separación es importante para entender la organización en capas de seguridad en los sistemas distribuidos y, en particular, la noción de confianza. La diferencia entre confianza y seguridad es importante. Un sistema es seguro o no (si se toman en cuenta varias medidas probabilísticas), pero que un cliente considere seguro a un sistema es una cuestión de confianza (Bishop, 2003). La seguridad es técnica; la confianza es emocional. La capa en que se colocan los mecanismos de seguridad depende de la confianza que un cliente tiene en qué tan seguros están los servicios en una capa particular.

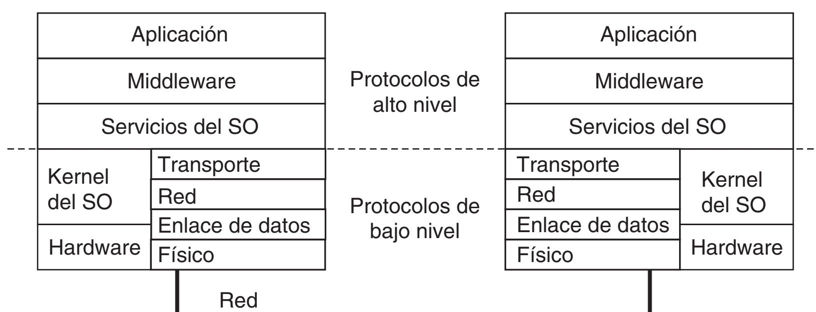


Figura 9-3. Organización lógica de un sistema distribuido en varias capas.

Como un ejemplo, consideremos una organización localizada en sitios diferentes que están conectados mediante un servicio de comunicación tal como **Switched Multi-megabit Data Service (SMDS).** Una red SMDS puede ser considerada como una red central (*backbone*) que conecta varias redes de área local ubicadas en sitios geográficos posiblemente dispersos, como se muestra en la figura 9-4.

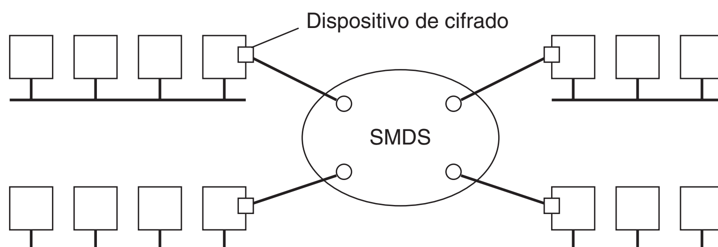


Figura 9-4. Varios sitios conectados mediante un servicio de conexión troncal de área amplia.

La seguridad puede proporcionarse colocando dispositivos de cifrado en cada ruteador SMDS, como también se muestra en la figura 9-4. Estos dispositivos cifran y descifran automáticamente los paquetes enviados entre sitios, aunque no proporcionan comunicación segura entre servidores en el mismo sitio. Si en el sitio *A* Alicia envía un mensaje a Bob en el sitio *B* y le preocupa que su mensaje sea interceptado, por lo menos debe confiar en que el cifrado del tráfico entre los sitios funciona apropiadamente. Esto significa, por ejemplo, que debe confiar en que los administradores de ambos sitios tomaron las medidas pertinentes contra la manipulación no autorizada de los dispositivos.

Ahora supongamos que Alicia no confía en la seguridad del tráfico entre sitios. Puede decidir entonces tomar sus propias medidas utilizando un servicio de seguridad a nivel de transporte, tal como SSL. SSL quiere decir **Secure Sockets Layer**, y puede ser utilizado para enviar mensajes con seguridad a través de una conexión TCP. Más adelante, en el capítulo 12, analizaremos los detalles

de un SSL cuando presentemos los sistemas basados en la web. El punto importante a observar en este caso es que el SSL permite a Alicia establecer una conexión segura con Bob. Todos los mensajes a nivel de transporte serán cifrados —y también al nivel SMDS, pero ello no preocupa a Alicia—. En este caso, tendrá que confiar en el SSL. En otras palabras, cree que SSL es seguro.

En sistemas distribuidos, los dispositivos de seguridad a menudo se colocan en la capa de middleware. Si Alicia no confía en el SSL, puede que desee utilizar un servicio RPC seguro. De nueva cuenta tendrá que confiar en que este servicio RPC cumple lo que promete, tal como nada de filtración de información o la autenticación apropiada de clientes y servidores.

Se puede confiar en los servicios de seguridad colocados en la capa de middleware sólo si los servicios de que dependen son realmente seguros. Por ejemplo, si se implementa un servicio RPC seguro en parte por medio de un SSL, entonces la confianza en el servicio RPC depende de cuánta confianza se tenga en el SSL. Si no se confía en el SSL, entonces no se puede confiar en la seguridad del servicio RPC.

Distribución de los mecanismos de seguridad

Las dependencias entre los servicios en cuanto a confianza conducen a la noción de una **Trusted Computing Base** (**TCB**). Una TCB es el conjunto de todos los mecanismos de seguridad implementados en un sistema de computadoras (distribuido) que son necesarios para hacer cumplir una política de seguridad, y que por tanto tiene que ser confiable. Mientras más pequeña sea la TCB, mejor. Si se construye un sistema distribuido como middleware en un sistema operativo de red existente, su seguridad depende de la seguridad de los sistemas operativos locales subyacentes. En otras palabras, en un sistema distribuido la TCB puede incluir los sistemas operativos locales en varios anfitriones.

Consideremos un servidor de archivos en un sistema distribuido. Tal servidor puede que requiera fiarse de los diversos mecanismos de protección ofrecidos por su sistema operativo local. Dichos mecanismos incluyen no sólo aquellos que protegen archivos contra accesos por parte de procesos distintos del servidor de archivos, sino también mecanismos que protegen el servidor contra intentos maliciosos de anularlo.

Los sistemas distribuidos basados en middleware requieren, por tanto, confiar en los sistemas operativos locales existentes de los que dependen. Si tal confianza no existe, entonces puede que requiera incorporar una parte de la funcionalidad de los sistemas operativos locales al propio sistema distribuido. Consideremos un sistema operativo de micronúcleo, en el cual la mayoría de sus servicios funcionan como procesos de usuario normales. En este caso, el sistema de archivos, por ejemplo, puede ser totalmente reemplazado por uno adecuado a las necesidades específicas de un sistema distribuido, incluidas sus diversas medidas de seguridad.

Compatible con este método es separar los servicios de seguridad de otros tipos de servicios distribuyéndolos en varias máquinas según el grado de seguridad requerido. Por ejemplo, para un sistema de archivos distribuido seguro, puede ser posible aislar el servidor de los clientes colocándolo en una máquina con un sistema operativo confiable, que posiblemente controle un sistema de archivos dedicado seguro. Los clientes y sus aplicaciones se colocan en máquinas no confiables.

Esta separación reduce en forma efectiva la TCB a un número relativamente pequeño de máquinas y componentes de software. Con la subsiguiente protección de dichas máquinas contra

ataques de seguridad externos, se puede incrementar la confianza total del sistema distribuido. En el método **Reduced Interfaces for Secure System Components (RISSC)** se impide el acceso directo de los clientes y sus aplicaciones a servicios críticos, tal como se describe en Neumann (1995). En el método RISSC, cualquier servidor crítico en cuanto a seguridad se coloca en otra máquina aislada de los sistemas de usuario final con interfaces de red de bajo nivel de seguridad, según muestra la figura 9-5. Los clientes y sus aplicaciones corren en máquinas diferentes y pueden acceder al servidor seguro sólo mediante estas interfaces de red.

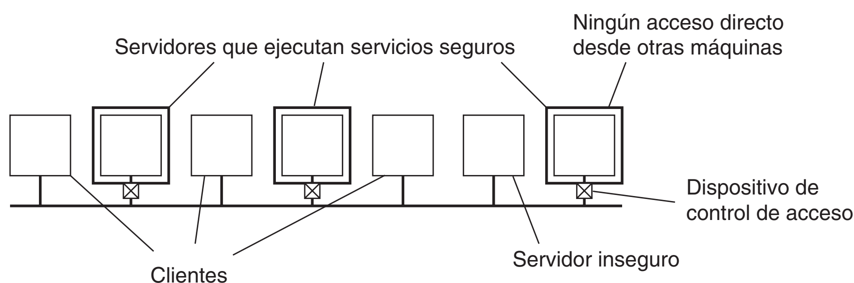


Figura 9-5. Principio del método RISSC tal como se aplica a sistemas distribuidos seguros.

Simplicidad

Otro tema de diseño importante relacionado con la decisión de en qué capa se colocarán los mecanismos de seguridad es el de simplicidad. El diseño de un sistema de computadora seguro se considera generalmente como una tarea difícil. Por consiguiente, si el diseñador de un sistema utiliza pocos mecanismos simples fáciles de entender y confiables de trabajar, mucho mejor.

Por desgracia, los mecanismos simples no siempre son suficientes para implementar políticas de seguridad. Volvamos a considerar la situación en que Alicia desea enviar un mensaje a Bob. El cifrado a nivel de vínculo es un mecanismo simple y fácil de entender que protege contra la intercepción del tráfico de mensajes entre sitios. No obstante, se necesita mucho más si Alicia quiere estar segura de que sólo Bob recibirá sus mensajes. En ese caso, se requieren servicios de autenticación a nivel de usuario, y puede que Alicia necesite saber cómo funcionan los servicios para confiar en ellos. La autenticación a nivel de usuario puede requerir, por consiguiente, cuando menos cierta noción acerca de claves criptográficas y algún conocimiento sobre mecanismos tales como los certificados, a pesar de que muchos servicios de seguridad son altamente automatizados y se encuentran ocultos para los usuarios.

En otros casos, la propia aplicación es inherentemente compleja y la introducción de seguridad sólo empeora las cosas. Un ejemplo de dominio de aplicación que implica protocolos de seguridad (como se verá más adelante en este capítulo) es el de sistemas de pago digitales. La complejidad de los protocolos de pago digitales a menudo se deriva de que múltiples personas necesitan comunicarse para realizar un pago. En estos casos, es importante que los mecanismos subyacentes utili-

zados para implementar los protocolos sean relativamente simples y fáciles de entender. La simplicidad contribuirá a incrementar la confianza que los usuarios finales depositarán en la aplicación y, aún más importante, convencerá a los diseñadores de que el sistema no tiene huecos de seguridad.

9.1.3 Criptografía

El uso de técnicas criptográficas es fundamental para la seguridad en los sistemas distribuidos. La idea básica de aplicar estas técnicas es simple. Consideremos un remitente *S* que desea transmitir un mensaje *m* a un destinatario *R*. Para proteger el mensaje contra amenazas de seguridad, el remitente primero lo **cifra** para transformarlo en un mensaje ininteligible *m′* y posteriormente envía *m′* a *R*. *R*, a su vez, debe **descifrar** el mensaje recibido para regresarlo a su forma original *m*.

El cifrado y el descifrado se logran con métodos criptográficos parametrizados por claves, como se muestra en la figura 9-6. La forma original del mensaje enviado se llama **texto común** (*plaintext*), mostrado como *P* en la figura 9-6; la forma cifrada se conoce como **texto cifrado** (*ciphertext*), ilustrado como *C*.

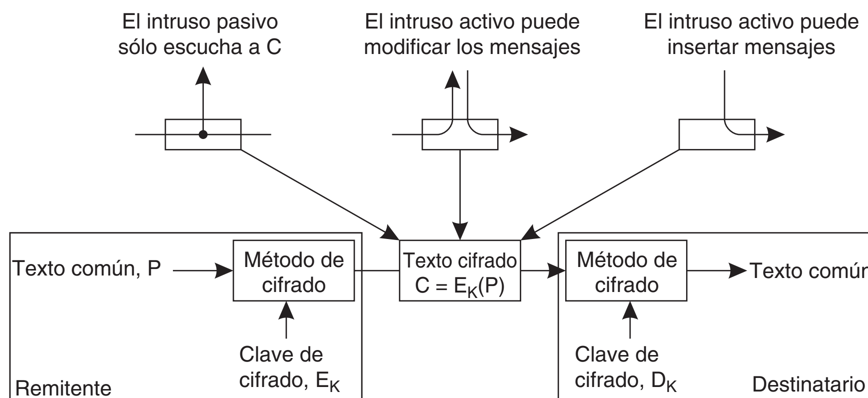


Figura 9-6. Intrusos y fisgones en comunicación.

Para describir los diversos protocolos de seguridad que se utilizan al construir servicios de seguridad para sistemas distribuidos, resulta muy útil contar con una notación que relacione el texto común, el texto cifrado y las claves. Siguiendo las convenciones de notación comunes, utilizaremos $C = E_K(P)$ para denotar que el texto cifrado *C* se obtiene cifrando el texto común *P* mediante la clave *K*. Asimismo, $P = D_K(C)$ es utilizada para expresar el descifrado del texto cifrado *C* por medio de la clave *K* para obtener el texto común *P*.

Volvamos al ejemplo mostrado en la figura 9-6, donde observamos que mientras se transfiere un mensaje como texto cifrado *C*, ocurren tres ataques contra los que hay que protegerse y para lo cual ayuda el cifrado. En primer lugar, un **intruso** puede interceptar el mensaje sin que el remitente

o el destinatario se den cuenta de que alguien está fisgoneando. Por supuesto, si el mensaje transmitido se cifró de tal suerte que no sea fácil de descifrar sin la clave apropiada, la intercepción es inútil; el intruso verá sólo datos ininteligibles. (A propósito, el solo hecho de transmitir un mensaje en ocasiones puede ser suficiente para que un intruso saque conclusiones. Por ejemplo, si durante una crisis mundial la cantidad de tráfico hacia la Casa Blanca se reduce fuertemente de improviso, en tanto que la cantidad de tráfico hacia cierta montaña de Colorado se incrementa en la misma cantidad, el hecho de saberlo puede ser información útil.)

El segundo tipo de ataque a ser tomado en cuenta es la modificación del mensaje. La modificación del texto común es fácil; la del texto que ha sido apropiadamente cifrado es mucho más difícil porque el intruso primero tendrá que descifrarlo antes de poder modificarlo. Además, también tendrá que volverlo a cifrar apropiadamente, de otro modo el destinatario podría advertir que el mensaje fue manipulado.

El tercer tipo de ataque es cuando un intruso inserta mensajes cifrados en el sistema de comunicación con el propósito de que R piense que estos mensajes fueron enviados por S. De nueva cuenta, como se verá más adelante en el capítulo, el cifrado puede proteger contra tales ataques. Observemos que si un intruso es capaz de modificar los mensajes, también es capaz de insertarlos.

Existe una distinción fundamental entre los diferentes sistemas criptográficos, la cual se basa en si las claves de cifrado y descifrado son o no las mismas. En un **sistema cifrado simétrico** se utiliza una misma clave para cifrar y descifrar un mensaje. En otros términos,

$$P = D_K(E_K(P))$$

Los sistemas cifrados simétricos también se conocen como sistemas de clave secreta o de clave compartida, porque se requiere que el remitente y el destinatario compartan la misma clave y, para asegurarse de que funcione la protección, esta clave compartida debe mantenerse secreta; a nadie más se le permite verla. Utilizaremos $K_{A,B}$ para denotar una clave compartida por A y B.

En un **sistema cifrado asimétrico**, las claves de cifrado y descifrado son diferentes, pero juntas forman un par único. En otros términos, existe una clave distinta K_E para cifrar y otra para descifrar, K_D, de modo que

$$P = D_{K_D}(E_{K_E}(P))$$

En un sistema cifrado asimétrico, una de las claves se mantiene privada; la otra se hace pública. Por esta razón, los sistemas cifrados asimétricos también se conocen como **sistemas de clave pública**. De aquí en adelante, usaremos la notación K_A^+ para denotar una clave pública perteneciente a A, y K_A^- como su clave privada correspondiente.

Anticipándonos a las detalladas explicaciones sobre protocolos de seguridad que presentaremos más adelante, cuál de las claves de cifrado y descifrado que en realidad se hace pública depende de cómo se utilicen las claves. Por ejemplo, si Alicia desea enviar un mensaje confidencial a Bob, deberá utilizar la clave pública de Bob para cifrar el mensaje. Como Bob es el único que conoce la clave de descifrado privada, él también es la única persona que puede descifrarlo.

Por otra parte, supongamos que Bob desea saber con seguridad que el mensaje recién recibido en realidad proviene de Alicia. En ese caso, Alicia puede mantener su clave de cifrado privada para cifrar los mensajes que envía. Si Bob es capaz de descifrar un mensaje con la clave pública de Alicia (el texto común en el mensaje contiene suficiente información significativa para Bob), sabrá que fue Alicia quien envió el mensaje porque la clave de descifrado está vinculada en forma única a la clave de cifrado. Más adelante volveremos a abordar con detalle tales algoritmos.

Una aplicación final de criptografía en sistemas distribuidos es el uso de **funciones hash**. Una función hash **H** considera un mensaje *m* de longitud arbitraria como entrada y produce una cadena de bits *h* de longitud fija como salida:

$$h = H(m)$$

Un hash *h* es comparable en cierta forma con los bits adicionales anexados a un mensaje en los sistemas de comunicación para permitir la detección de errores, tal como la verificación de redundancia cíclica (*CRC*, por sus siglas en inglés).

Las funciones hash utilizadas en sistemas criptográficos tienen varias propiedades esenciales. En primer lugar, son **funciones unidireccionales**, es decir, computacionalmente no es factible encontrar la entrada *m* que corresponda a una salida conocida *h*. Por otra parte, calcular *h* a partir de *m* es fácil. En segundo lugar, las funciones unidireccionales tienen la propiedad de **resistencia débil a la colisión**, esto significa que, dada una entrada *m* y su salida asociada $h = H(m)$, computacionalmente no es factible ubicar otra entrada diferente $m' \neq m$ de modo que $H(m) = H(m')$. Por último, las funciones hash criptográficas también tienen la propiedad de **resistencia fuerte a la colisión**, ello significa que, cuando sólo *H* es conocida, computacionalmente no es factible hallar dos valores de entrada diferentes *m* y *m′*, así que $H(m) = H(m')$.

La función de cifrado *E* y las claves utilizadas deben tener propiedades similares. Además, para cualquier función de cifrado *E*, no deberá ser computacionalmente factible hallar la clave *K* cuando se proporciona el texto común *P* y el texto cifrado asociado $C = E_K(P)$. Asimismo, análogo a la resistencia a la colisión, cuando se tiene un texto común *P* y una clave *K*, deberá ser efectivamente imposible encontrar otra clave *K′* de modo que $E_K(P) = E_{K'}(P)$.

El arte y la ciencia de concebir algoritmos para sistemas criptográficos tiene una larga y fascinante historia (Kahn, 1967), y la construcción de sistemas seguros a menudo es sorprendentemente difícil o incluso imposible (Schneier, 2000). Queda fuera del alcance de este libro estudiar cualquiera de tales algoritmos con detalle. Sin embargo, para tener una idea de criptografía en sistemas de computadora, a continuación se analizarán brevemente tres algoritmos representativos. Información detallada sobre éstos y otros algoritmos criptográficos se localiza en Ferguson y Schneier (2003), Menezes y colaboradores (1996), y Schneier (1996).

Antes de entrar en detalles de los diversos protocolos, en la figura 9-7 se presenta un resumen de la notación y las abreviaturas que se utilizan en las expresiones matemáticas siguientes.

Criptosistemas simétricos: DES

Un primer ejemplo de un algoritmo criptográfico es el **Data Encryption Standard** (**DES**), el cual se utiliza para implementar criptosistemas simétricos. DES se diseñó para operar con bloques

Notación	Descripción
$K_{A,B}$	Clave secreta compartida por *A* y *B*
K_A^+	Clave pública de *A*
K_A^-	Clave privada de *A*

Figura 9-7. Notación utilizada en este capítulo.

de datos de 64 bits. Un bloque se transforma en un bloque de salida cifrado (64 bits) en 16 rondas, donde cada ronda utiliza una clave diferente de 48 bits de cifrado. Cada una de estas 16 claves se deriva de una clave maestra de 56 bits, como se muestra en la figura 9-8(a). Antes de que un bloque de entrada inicie sus 16 rondas de cifrado, primero se somete a una permutación inicial, de la cual posteriormente se aplica la inversa a la salida cifrada que conduce al bloque de salida final.

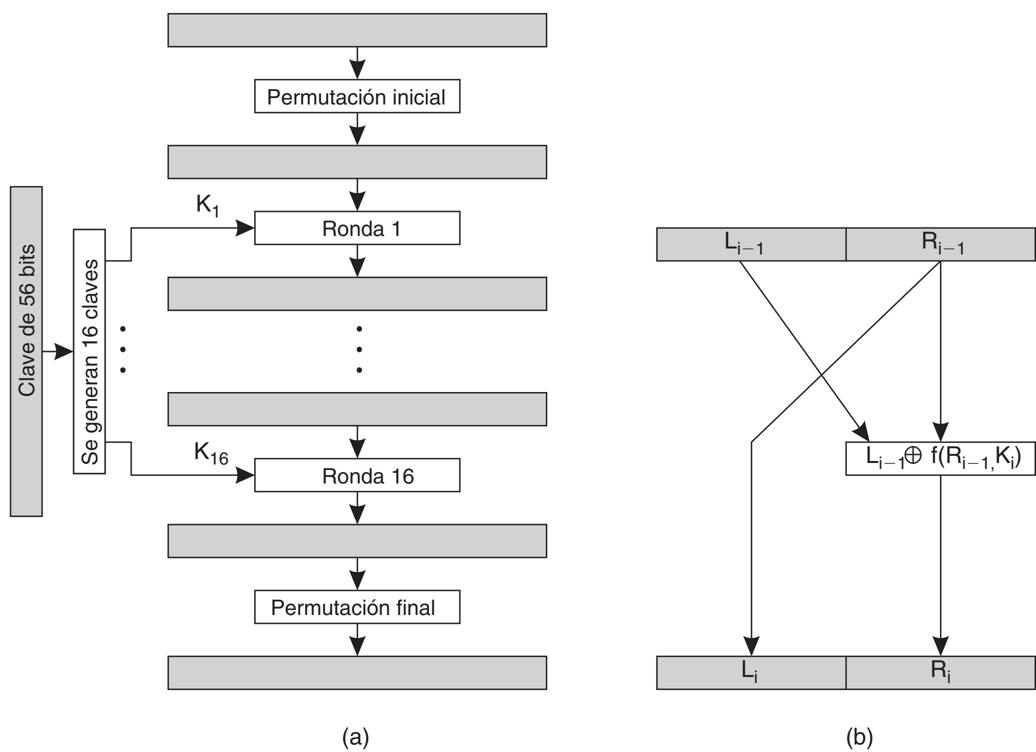


Figura 9-8. (a) Principio del algoritmo DES. (b) Plan general de una ronda de cifrado.

Cada ronda de cifrado *i* toma el bloque de 64 bits producido por la ronda previa $i - 1$ como su entrada, según muestra la figura 9-8(b). Los 64 bits se dividen en una parte izquierda L_{i-1} y una parte derecha R_{i-1}, cada parte consta de 32 bits. La parte derecha se utiliza para la parte izquierda en la siguiente ronda, es decir, $L_i = R_{i-1}$.

El trabajo duro se realiza en la función mangler *f*. Esta función toma un bloque de 32 bits R_{i-1} como entrada, junto con una clave de K_i de 48 bits, y produce un bloque de 32 bits al que se le aplica la función XOR con L_{i-1} para producir R_i. (XOR es una abreviatura de la operación excluyente o.) La función mangler primero expande R_{i-1} a un bloque de 48 bits y le aplica la función XOR con K_i. El resultado se divide en ocho secciones de seis bits cada una. Cada sección se alimenta hacia una **caja S** diferente, la cual es una operación que sustituye cada una de las 64 entradas de seis bits posibles en una de las 16 salidas de 4 bits posibles. Las ocho secciones de salida de cuatro bits cada una se combinan entonces en un valor de 32 bits y se vuelven a permutar.

La clave K_i de 48 bits para la ronda *i* se deriva de la clave maestra de 56 bits como sigue. En primer lugar, la clave maestra se permuta y divide en mitades de 28 bits. Para cada ronda, cada mitad se rota primero uno o dos bits a la izquierda, tras de lo cual se extraen 24 bits. Junto con los 24 bits de la otra mitad rotada, se construye una clave de 48 bits. Los detalles de una ronda de cifrado se muestran en la figura 9-9.

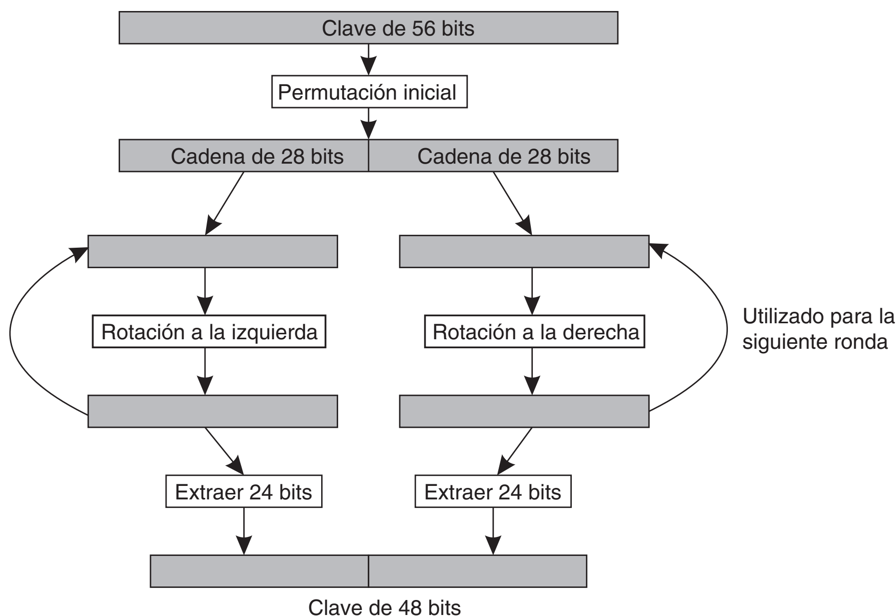


Figura 9-9. Detalles de la generación de claves por cada ronda en DES.

El principio de DES resulta bastante simple, pero el algoritmo es difícil de romper mediante métodos analíticos. El uso de un ataque de fuerza bruta de simplemente buscar una clave que realice el trabajo se ha vuelto fácil, tal como se ha demostrado en varias ocasiones. No obstante, aplicar DES tres veces en un modo especial de cifrar-descifrar-cifrar con claves diferentes, proce-

dimiento conocido también como **Triple DES**, es mucho más seguro y a menudo es el que se utiliza [vea también Barker (2004)].

El que DES sea difícil de atacar mediante análisis es porque el razonamiento de diseño subyacente nunca ha sido explicado en público. Por ejemplo, se sabe que si se toman otras cajas S diferentes de las actualmente utilizadas en el estándar, el algoritmo se vuelve sustancialmente más fácil de violar (vea Pfleeger, 2003, para conocer un breve análisis de DES). Un razonamiento para el diseño y uso de las cajas S se publicó sólo después de que se idearon nuevos modelos de ataque en los años de 1990. DES demostró ser bastante resistente a estos ataques y sus diseñadores revelaron que los modelos recién ideados ya les eran conocidos cuando desarrollaron DES en 1974 (Coppersmith, 1994).

DES ha sido utilizado como técnica de cifrado estándar por años, pero en la actualidad se encuentra en proceso de ser reemplazado por los bloques de algoritmo Rijndael de 128 bits. También existen variantes con claves y bloques de datos más grandes. El algoritmo se diseñó para que fuera lo suficientemente rápido como para ser implementado en tarjetas inteligentes, lo cual constituye un área de aplicación cada vez más importante para la criptografía.

Criptosistemas de clave pública: RSA

El segundo ejemplo de algoritmo criptográfico es ampliamente utilizado en sistemas de clave pública: **RSA**, nombrado así en honor de sus inventores Rivest, Shamir y Adleman (1978). La seguridad de RSA se deriva de que no se conocen métodos para calcular con eficiencia los **factores primos** de números grandes. Es posible demostrar que cada entero puede ser escrito como el producto de números primos. Por ejemplo, 2100 se puede escribir como

$$2100 = 2 \times 2 \times 3 \times 5 \times 5 \times 7$$

ello hace que 2, 3, 5 y 7 sean los factores primos de 2100. En RSA, las claves privadas y públicas se construyen con números primos muy grandes (compuestos por cientos de dígitos decimales). Como se observa, violar RSA equivale a encontrar esos números primos. Hasta ahora, se ha demostrado que esto no es computacionalmente factible a pesar de que los matemáticos han tratado de resolver el problema durante siglos.

La generación de las claves privadas y públicas requiere cuatro pasos:

1. Elegir dos números primos muy grandes, p y q.
2. Calcular $n = p \times q$ y $z = (p - 1) \times (q - 1)$.
3. Elegir un número d que sea relativamente primo de z.
4. Calcular el número e de modo que $e \times d = 1 \; mod \; z$.

Uno de los números, por ejemplo d, puede ser utilizado posteriormente para descifrar, en tanto que e se utiliza para cifrar. Sólo uno de estos dos se hace público, según para lo que se esté utilizando el algoritmo.

Consideremos el caso en que Alicia desea mantener confidenciales los mensajes que envía a Bob. Es decir, quiere estar segura de que nadie excepto Bob intercepte y lea sus mensajes. RSA considera que cada mensaje m es una cadena de bits. Cada mensaje se divide primero en bloques de longitud fija, donde cada bloque m_i, interpretado como un número binario, deberá quedar en el intervalo $0 \leq m_i < n$.

Para cifrar el mensaje m, el remitente calcula para cada bloque m_i el valor $c_i = m_i^e \ (mod\ n)$, que es enviado entonces al destinatario. El descifrado del lado del destinatario ocurre calculando $m_i = c_i^d \ (mod\ n)$. Observemos que para implementar el cifrado se requieren tanto e como n, mientras que el descifrado requiere conocer los valores de d y n.

Cuando se compara RSA con criptosistemas simétricos tales como DES, RSA tiene la desventaja de ser computacionalmente más complejo. Por tanto, el cifrado de mensajes con RSA es aproximadamente 100-1000 veces más lento que DES, según la técnica de implementación utilizada. Por consiguiente, muchos sistemas criptográficos utilizan RSA para intercambiar sólo claves compartidas en una forma segura, pero no para cifrar datos "normales". En secciones subsiguientes se verán ejemplos de la combinación de estas dos técnicas.

Funciones hash: MD5

Como último ejemplo de un algoritmo criptográfico ampliamente utilizado, daremos un vistazo a MD5 (Rivest, 1992). **MD5** es una función hash útil para calcular un **resumen de mensaje** de longitud fija de 128 bits tomado de una cadena de entrada binaria de longitud arbitraria. La cadena de entrada se rellena primero hasta una longitud total de 448 bits (módulo 512), después de lo cual la longitud de la cadena de bits original se rellena con un entero de 64 bits. En realidad, la entrada se convierte en una serie de bloques de 512 bits.

La estructura del algoritmo se muestra en la figura 9-10. Comenzando con algún valor constante de 128 bits, el algoritmo prosigue en k fases, donde k es el número de bloques de 512 bits que comprende el mensaje rellenado. Durante cada fase, se calcula un resumen de 128 bits de un bloque de datos de 512 bits tomado del mensaje rellenado y del resumen de 128 bits calculado en la fase precedente.

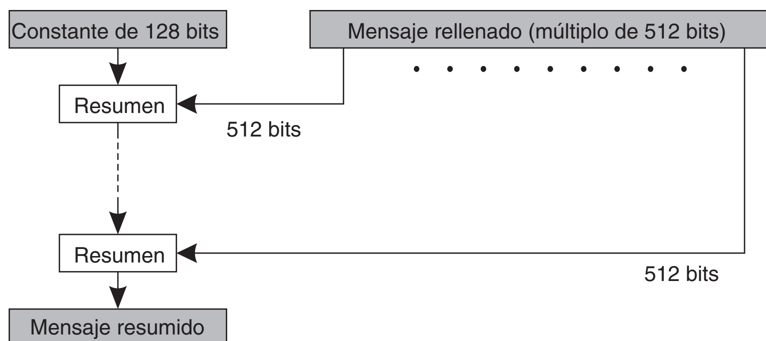


Figura 9-10. Estructura de una función MD5.

En MD5, una fase consta de cuatro rondas de cálculos, donde cada ronda utiliza una de las siguientes cuatro funciones:

$F(x,y,z) = (x\ AND\ y)\ OR\ ((NOT\ x)\ AND\ z)$
$G(x,y,z) = (x\ AND\ z)\ OR\ ((y\ AND\ (NOT\ z))$
$H(x,y,z) = x\ XOR\ y\ XOR\ z$
$I(x,y,z) = y\ XOR\ (x\ OR\ (NOT\ z))$

Cada una de estas funciones opera con variables de 32 bits *x*, *y*, y *z*. Para ilustrar cómo se utilizan estas funciones, consideremos un bloque de 512 bits *b* del mensaje incrustado que está siendo procesado durante la fase *k*. El bloque *b* se divide en bloques secundarios (o subbloques) de 32 bits b_0, $b_1,\ldots, b_{15}$. Durante la primera ronda, se utiliza la función *F* para cambiar cuatro variables (denotadas como *p*, *q*, *r*, y *s*, respectivamente) en 16 iteraciones, según muestra la figura 9-11. Estas variables se arrastran hacia cada una de las siguientes rondas, y una vez que se termina una fase, se pasan a la siguiente fase. Existe un total de 64 constantes predefinidas C_i. La notación $x \lll n$ se utiliza para indicar una *rotación izquierda*: los bits de *x* se desplazan *n* posiciones a la izquierda, donde el bit desplazado desde la izquierda se coloca en la posición localizada más a la derecha.

Iteraciones 1 a 8	**Iteraciones 9 a 16**
$p \leftarrow (p + F(q,r,s) + b_0 + C_1) \lll 7$	$p \leftarrow (p + F(q,r,s) + b_8 + C_9) \lll 7$
$s \leftarrow (s + F(p,q,r) + b_1 + C_2) \lll 12$	$s \leftarrow (s + F(p,q,r) + b_9 + C_{10}) \lll 12$
$r \leftarrow (r + F(s,p,q) + b_2 + C_3) \lll 17$	$r \leftarrow (r + F(s,p,q) + b_{10} + C_{11}) \lll 17$
$q \leftarrow (q + F(r,s,p) + b_3 + C_4) \lll 22$	$q \leftarrow (q + F(r,s,p) + b_{11} + C_{12}) \lll 22$
$p \leftarrow (p + F(q,r,s) + b_4 + C_5) \lll 7$	$p \leftarrow (p + F(q,r,s) + b_{12} + C_{13}) \lll 7$
$s \leftarrow (s + F(p,q,r) + b_5 + C_6) \lll 12$	$s \leftarrow (s + F(p,q,r) + b_{13} + C_{14}) \lll 12$
$r \leftarrow (r + F(s,p,q) + b_6 + C_7) \lll 17$	$r \leftarrow (r + F(s,p,q) + b_{14} + C_{15}) \lll 17$
$q \leftarrow (q + F(r,s,p) + b_7 + C_8) \lll 22$	$q \leftarrow (q + F(r,s,p) + b_{15} + C_{16}) \lll 22$

Figura 9-11. Las 16 iteraciones presentes durante la primera ronda en una fase de MD5.

De igual modo, la segunda ronda utiliza la función *G*, en tanto que *H* e *I* se utilizan en la tercera y cuarta rondas, respectivamente. Cada paso consta, por tanto, de 64 iteraciones, después de lo cual se inicia la siguiente fase, pero ahora con los valores que *p*, *q*, *r*, y *s* tienen en ese punto.

9.2 CANALES SEGUROS

En los capítulos precedentes, se utilizó con frecuencia el modelo cliente-servidor como una forma conveniente de organizar un sistema distribuido. En este modelo, los servidores posiblemente pueden ser distribuidos o replicados, aunque también actúan como clientes con respecto a los demás

servidores. Cuando se considera la seguridad en sistemas distribuidos, una vez más resulta muy útil pensar en función de clientes y servidores. En particular, implementar un sistema distribuido seguro se reduce esencialmente a dos temas predominantes. El primero es cómo hacer segura la comunicación entre clientes y servidores. La comunicación segura requiere autenticación de las partes que se comunican. En muchos casos, también es necesario garantizar la integridad y posiblemente la confidencialidad de un mensaje. Como parte de este problema, se debe considerar además la protección de la comunicación dentro de un grupo de servidores.

El segundo tema es el de autenticación; una vez que un servidor ha aceptado la solicitud de un cliente, ¿cómo puede saber si dicho cliente está autorizado para realizar la solicitud? La autorización tiene que ver con el problema de controlar el acceso a recursos, lo cual se analiza extensamente en la siguiente sección. Esta sección se concentra en la protección de la comunicación dentro de un sistema distribuido.

El tema de la protección de la comunicación entre clientes y servidores puede ser considerado en función del establecimiento de un **canal seguro** entre las partes que se están comunicando (Voydock y Kent, 1983). Un canal seguro protege a remitentes y destinatarios contra intercepción, modificación, y fabricación de mensajes. No necesariamente protege además contra interrupción. La protección de mensajes contra intercepción se realiza garantizando la confidencialidad: el canal seguro garantiza que sus mensajes no pueden ser vistos por intrusos. La protección contra modificación y fabricación se realiza mediante protocolos de autenticación mutua e integridad del mensaje. En las páginas siguientes, se analizan primero varios protocolos que pueden ser utilizados para implementar la autenticación por medio de criptosistemas simétricos y de clave pública. Una descripción detallada de la lógica que sirve de fundamento para la autenticación se encuentra en Lampson y colaboradores (1992). La confidencialidad y la integridad de los mensajes se estudian por separado.

9.2.1 Autenticación

Antes de entrar en los detalles de varios protocolos de autenticación, vale la pena señalar que la autenticación y la integridad de los mensajes no se pueden realizar una sin la otra. Consideremos, por ejemplo, un sistema distribuido que soporta la autenticación de dos partes en comunicación, pero que no proporciona mecanismos para garantizar la integridad del mensaje. En un sistema como ese, Bob puede saber con seguridad que Alicia es el remitente de un mensaje m. Sin embargo, si a Bob no se le puede garantizar que m no ha sido modificado durante la transmisión, ¿de qué le sirve saber que Alicia envió (la versión original de) m?

Asimismo, supongamos que únicamente la integridad del mensaje es soportada, pero que no existen mecanismos de autenticación. Cuando Bob reciba un mensaje para informarle que se ganó $1 000 000 en la lotería, ¿qué tan feliz podría estar si no puede verificar si el mensaje fue enviado por el organizador de dicha lotería?

Por consiguiente, la autenticación y la integridad del mensaje deben ir de la mano. En muchos protocolos, la combinación funciona aproximadamente como sigue. De nueva cuenta, supongamos que Alicia y Bob desean comunicarse y que Alicia toma la iniciativa al establecer un canal. Alicia envía un mensaje a Bob, o de lo contrario a una tercera parte confiable que le ayudará a establecer

el canal. Una vez que el canal ha sido establecido, Alicia sabe que está hablando con Bob y éste sabe con toda seguridad que está hablando con Alicia, así que pueden intercambiar mensajes.

Para garantizar la posterior integridad de los mensajes de datos intercambiados una vez autentificados, es práctica común utilizar criptografía de clave secreta por medio de claves de sesión. Una **clave de sesión** es una clave (secreta) compartida que se utiliza para cifrar mensajes en cuanto a integridad y, posiblemente, en cuanto a confidencialidad. Tal clave se usa generalmente sólo en tanto el canal existe. Cuando éste se cierra, su clave de sesión asociada se desecha (o en realidad, seguro se destruye). En seguida regresamos a las claves de sesión.

Autenticación basada en una clave secreta compartida

Primero daremos un vistazo a un protocolo de autenticación basado en una clave secreta que ya comparten Alicia y Bob. Cómo se las arreglan en realidad para obtener una clave compartida de una manera segura se analiza más adelante en este capítulo. En la descripción del protocolo, Alicia y Bob se abrevian A y B, respectivamente, y su clave compartida es denotada como $K_{A,B}$. El protocolo adopta un método común mediante el cual una parte reta a la otra a que responda correctamente sólo si la otra parte conoce la clave secreta compartida. Tales soluciones también se conocen como **protocolos de reto-respuesta**.

En el caso de autenticación basada en una clave secreta compartida, el protocolo continúa como se muestra en la figura 9-12. Primero, Alicia envía su identidad a Bob (mensaje 1), lo cual indica que desea establecer un canal de comunicación entre los dos. Bob envía posteriormente un reto R_B a Alicia, mostrado como el mensaje 2. Tal reto podría adoptar la forma de un número aleatorio. Se requiere que Alicia cifre el reto con la clave secreta $K_{A,B}$ que comparte con Bob, y lo regrese cifrado a Bob. Esta respuesta se muestra como el mensaje 3 en la figura 9-12 que contiene $K_{A,B}(R_B)$.

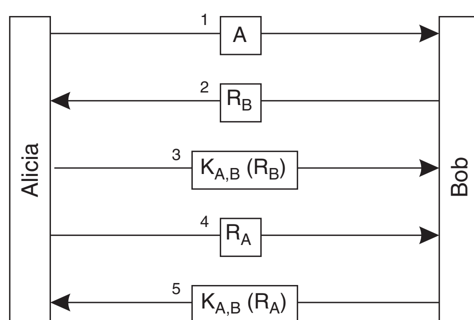


Figura 9-12. Autenticación basada en una clave secreta compartida.

Cuando Bob recibe la respuesta $K_{A,B}(R_B)$ a su reto R_B, puede descifrar el mensaje volviendo a utilizar la clave compartida para ver si contiene R_B. De ser así, entonces sabe que Alicia está del otro lado, ¿pues quién más pudo haber descifrado R_B con $K_{A,B}$? En otros términos, Bob ya comprobó

que sí está hablando con Alicia. No obstante, observemos que Alicia aún no ha verificado si es Bob quien está del otro lado del canal. Así, envía un reto R_A (mensaje 4), al cual Bob responde regresando $K_{A,B}(R_A)$, mostrado como el mensaje 5. Cuando Alicia lo descifra con $K_{A,B}$ y ve su R_A, se da cuenta de que está hablando con Bob.

Uno de los temas de seguridad más difíciles al diseñar protocolos es que realmente funcionen. Para ilustrar cuán fácilmente se pueden complicar las cosas, consideremos una "optimización" del protocolo de autenticación en el cual el número de mensajes se ha reducido de cinco a tres, como se muestra en la figura 9-13. La idea básica es que si Alicia finalmente desea retar a Bob, deberá enviarle un reto junto con su identidad cuando establezca el canal. Asimismo, Bob responde al reto, junto con su propio reto en un solo mensaje.

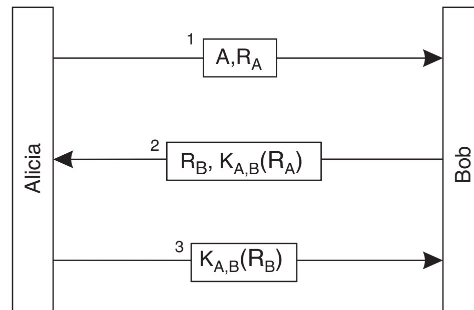


Figura 9-13. Autenticación basada en una clave secreta compartida, pero utilizando tres mensajes en lugar de cinco.

Por desgracia, este protocolo ya no funciona. Puede ser derrotado con facilidad por lo que se conoce como un **ataque de reflexión**. Para explicar cómo funciona dicho ataque, consideremos un intruso llamado Chuck, al que se denota como C en los protocolos. El objetivo de Chuck es establecer un canal con Bob de modo que éste crea que está hablando con Alicia. Chuck puede hacerlo si responde correctamente al reto enviado por Bob, por ejemplo, regresando la versión cifrada de un número que Bob envió. Sin conocimiento de $K_{A,B}$, sólo Bob puede realizar el descifrado y, precisamente, lo que Chuck pretende es engañar a Bob para que lo haga.

El ataque se ilustra en la figura 9-14. Chuck envía un mensaje que contiene la identidad de Alicia A, junto con un reto R_C. Bob regresa el reto R_B y la respuesta $K_{A,B}(R_C)$ en un solo mensaje. En ese momento, Chuck tendría que comprobar que conoce la clave secreta regresando $K_{A,B}(R_B)$ a Bob. Desafortunadamente, no cuenta con $K_{A,B}$. Entonces, intenta establecer un segundo canal para hacer que Bob realice el descifrado por él.

Por consiguiente, Chuck envía a A y R_B en un solo mensaje como antes, pero ahora da a entender que desea un segundo canal. Esto se muestra como el mensaje 3 en la figura 9-14. Bob, al no darse cuenta que él, él mismo, había utilizado R_B antes como reto, responde $K_{A,B}(R_B)$ y envía otro reto R_{B2}, mostrado como el mensaje 4. En ese punto, Chuck ya cuenta con $K_{A,B}(R_B)$ y termina iniciando

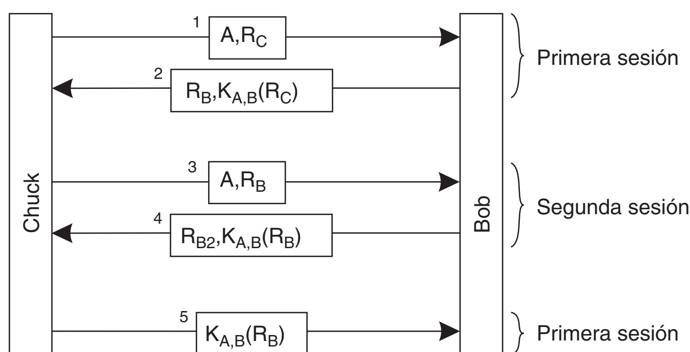


Figura 9-14. El ataque de reflexión.

la primera sesión al regresar el mensaje 5 que contiene la respuesta $K_{A,B}(R_B)$, la cual originalmente fue solicitada con el reto enviado en el mensaje 2.

Tal como se explica en Kaufman y colaboradores (2003), uno de los errores cometidos durante la adaptación del protocolo original fue que en la nueva versión del protocolo las dos partes estuvieron utilizando el mismo reto en dos ejecuciones diferentes del protocolo. Un mejor diseño es utilizar siempre retos diferentes tanto para quien inicia como para quien responde. Por ejemplo, si Alicia utiliza siempre un número impar y Bob uno par, Bob se habría dado cuenta de que algo raro estaba ocurriendo cuando recibió el R_B en el mensaje 3 en la figura 9-14. (Desafortunadamente, esta solución está sujeta a otros ataques, de modo notable, a uno conocido como "hombre en medio del ataque", el cual se explica en Ferguson y Schneier, 2003.) En general, permitir que las dos partes que están estableciendo un canal realicen varias cosas de forma idéntica no es una buena idea.

Otro principio que se viola en el protocolo adaptado es que Bob liberó información valiosa en la forma de la respuesta $K_{A,B}(R_C)$ sin estar seguro de a quién se la estaba dando. Este principio no fue violado en el protocolo original, donde Alicia tiene que demostrar primero su identidad, tras de lo cual Bob deseaba pasarle su información cifrada.

Existen otros principios que los desarrolladores de protocolos criptográficos han aprendido gradualmente al paso de los años, y se presentarán algunos cuando se analicen otros protocolos más adelante. Una lección importante es que diseñar protocolos de seguridad que realicen lo que se supone deben realizar a menudo es mucho más difícil de lo que parece. Asimismo, el más leve ajuste de un protocolo existente, para mejorar su desempeño, puede afectar con facilidad su exactitud, tal como ya se demostró. Más teoría sobre principios de diseño de protocolos se encuentra en Abadi y Needham (1996).

Autenticación mediante un centro de distribución de clave

Uno de los problemas que se presentan con el uso de una clave secreta compartida para autenticación es su escalabilidad. Si un sistema distribuido contiene N servidores, y cada servidor requiere

compartir una clave secreta con los demás $N - 1$ servidores, el sistema en su conjunto necesita manejar $N(N - 1)/2$ claves, y cada servidor debe manejar $N - 1$ claves. Con N grande, esto conduce a problemas. Una alternativa es utilizar un método centralizado por medio de un **centro de distribución de clave** (**KDC**, por sus siglas en inglés). Este KDC comparte una clave secreta con cada uno de los servidores, pero no se requiere que ningún par de ellos tenga también una clave secreta compartida. En otros términos, el uso de un KDC requiere que se manejen N claves en lugar de $N(N - 1)/2$, lo cual claramente es una mejora.

Si Alicia desea establecer un canal seguro con Bob, puede hacerlo con ayuda de un KDC confiable. La idea integral es que el KDC entregue una clave tanto a Alicia como a Bob que puedan utilizar para comunicarse, y se muestra en la figura 9-15.

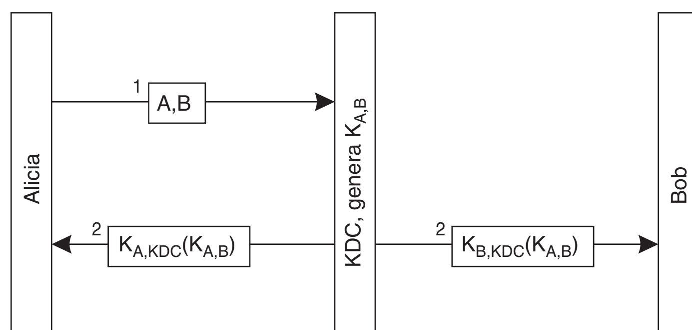


Figura 9-15. El principio de utilizar un KDC.

Alicia envía primero un mensaje al KDC, informándole que desea hablar con Bob. El KDC regresa el mensaje conteniendo una clave secreta compartida $K_{A,B}$ que Alicia pueda utilizar. El mensaje se cifra con la clave secreta $K_{A,KDC}$ que Alicia comparte con el KDC. Además, el KDC también envía $K_{A,B}$ a Bob, pero ahora cifrada con la clave secreta $K_{B,KDC}$ que comparte con Bob.

La desventaja principal de este método es que puede suceder que Alicia desee establecer un canal seguro con Bob incluso antes de que Bob haya recibido la clave compartida del KDC. Además, se requiere que el KDC integre a Bob al lazo pasándole la clave. Estos problemas pueden ser evitados si el KDC regresa $K_{B,KDC}(K_{A,B})$ a Alicia y le permite conectarse con Bob. Esto conduce al protocolo mostrado en la figura 9-16. El mensaje $K_{B,KDC}(K_{A,B})$ también se conoce como **boleto**. Es tarea de Alicia pasar este boleto a Bob. Observemos que Bob sigue siendo el único que puede hacer un uso razonable del boleto, ya que es el único además del KDC que sabe cómo descifrar la información que contiene.

El protocolo mostrado en la figura 9-16 es en realidad una variante de un ejemplo muy conocido de un protocolo de autenticación que utiliza un KDC, denominado **protocolo de autenticación de Needham-Schroeder** en honor de sus inventores (Needham y Schroeder, 1978). Una variante diferente del protocolo se utiliza en el sistema Kerberos, la cual describimos más adelante. El protocolo Needham-Schroeder, mostrado en la figura 9-17, es un protocolo de reto-respuesta multidireccional que funciona como sigue.

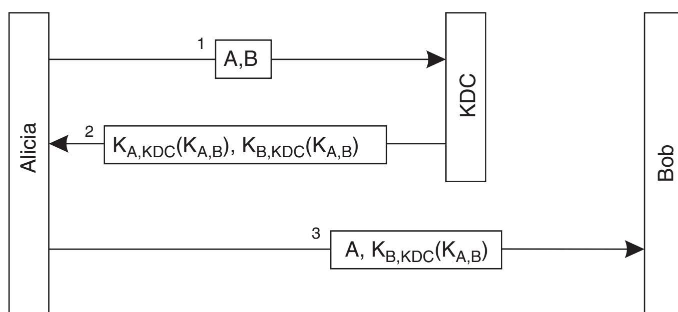


Figura 9-16. Utilización de un boleto y permiso para que Alicia se conecte con Bob.

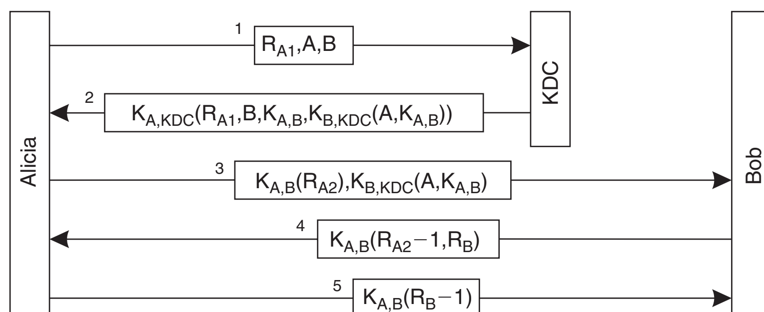


Figura 9-17. Protocolo de autenticación de Needham-Schroeder.

Cuando Alicia desea establecer un canal seguro con Bob, envía una petición al KDC que contiene un reto R_A, junto con su identidad A y, por supuesto, la de Bob. El KDC responde otorgándole el boleto $K_{B,KDC}$ $(K_{A,B})$ junto con la clave secreta $K_{A,B}$ que posteriormente, puede compartir con Bob.

El reto R_{A1} que Alicia envía al KDC junto con su respuesta para establecer un canal con Bob también se conoce como nonce. Un **nonce** es un número aleatorio que sólo se utiliza una vez, tal como un número elegido a partir de un conjunto muy grande. El propósito principal de un nonce es relacionar en forma única dos mensajes, en este caso el mensaje 1 y el mensaje 2. En particular, con R_{A1} incluido otra vez en el mensaje 2, Alicia sabrá con certeza que el mensaje 2 es enviado como respuesta al mensaje 1, y que no es, por ejemplo, un reenvío de un mensaje más viejo.

Para entender este problema, supongamos que no se utilizaron nonces y que Chuck se robó una de las viejas claves de Bob, por ejemplo $K_{B,KDC}^{vieja}$. Además, Chuck interceptó una respuesta vieja $K_{A,KDC}(B,K_{A,B},K_{B,KDC}^{vieja}(A,K_{A,B}))$ que el KDC había devuelto en respuesta a una petición previa de Alicia de hablar con Bob. Mientras tanto, Bob tendrá que negociar una nueva clave

secreta compartida con el KDC. Sin embargo, Chuck espera pacientemente hasta que Alicia vuelve a solicitar establecer un canal seguro con Bob. En ese momento, envía la respuesta vieja y hace que Alicia crea que está hablando con Bob, porque puede descifrar el boleto y demostrar que conoce la clave secreta compartida $K_{A,B}$. Claramente esto es inaceptable y habrá que tomar medidas en su contra.

Si se incluye un nonce, tal ataque es imposible porque el reenvío de un mensaje viejo será descubierto de inmediato. En particular, el nonce incluido en el mensaje de respuesta no coincidirá con el nonce de la petición original.

El mensaje 2 también contiene *B*, la identidad de Bob. Al incluir *B*, el KDC protege a Alicia contra el siguiente ataque. Supongamos que *B* fue dejada afuera del mensaje 2. En ese caso, Chuck podría modificar el mensaje 1 reemplazando la identidad de Bob con su propia identidad, es decir, *C*. El KDC pensaría entonces que Alicia desea establecer un canal seguro con Chuck y responde en la forma apropiada. En cuanto Alicia desea ponerse en contacto con Bob, Chuck intercepta el mensaje y hace creer a Alicia que está hablando con Bob. Al copiar la identidad de la otra parte del mensaje 1 en el mensaje 2, Alicia de inmediato detectará que su petición ha sido modificada.

Una vez que el KDC pasa el boleto a Alicia, el canal seguro entre Alicia y Bob puede ser establecido. Alicia envía el mensaje 3, el cual contiene el boleto para Bob y un reto R_{A2} cifrado con la tecla compartida $K_{A,B}$ que el KDC acaba de generar. Bob descifra entonces el boleto para descubrir la clave compartida y regresa una respuesta $R_{A2} - 1$ junto con el reto R_B para Alicia.

Se impone el siguiente comentario con respecto al mensaje 4. En general, regresando $R_{A2} - 1$ y no solamente R_{A2}, Bob no sólo demuestra que conoce la clave secreta compartida, sino también que en realidad descifró el reto. De nueva cuenta, esto vincula el mensaje 4 al 3 del mismo modo que R_A vinculó el mensaje 2 al 1. El protocolo, por tanto, está más protegido contra reenvíos.

Sin embargo, en este caso especial, habría sido suficiente con regresar $K_{A,B}(R_{A2}, R_B)$, por la simple razón de que este mensaje nunca antes ha sido utilizado en el protocolo. $K_{A,B}(R_{A2},R_B)$ ya demostró que Bob ha sido capaz de descifrar el reto enviado en el mensaje 3. El mensaje 4 como lo muestra la figura 9-17 se debe a razones históricas.

Tal como se presenta aquí, el protocolo Needham-Schroeder sigue teniendo el punto débil de que si Chuck llega a hacerse de una vieja clave $K_{A,B}$, podría reenviar el mensaje 3 y conseguir que Bob establezca un canal. Bob creerá entonces que está hablando con Alicia cuando, en realidad, Chuck está en el otro extremo. En este caso, se tiene que relacionar el mensaje 3 con el mensaje 1, es decir, hacer que la clave dependa de la petición inicial de Alicia de establecer un canal con Bob. La solución se muestra en la figura 9-18.

El truco es incorporar un nonce a la petición enviada por Alicia al KDC. Sin embargo, el nonce tiene que venir de Bob; esto hace que Bob se sienta seguro de que quienquiera que desee establecer un canal seguro con él, habrá obtenido la información apropiada del KDC. Por consiguiente, Alicia primero solicita a Bob que le envíe un nonce R_{B1} cifrado con la clave compartida entre Bob y el KDC. Alicia incorpora este nonce a su petición al KDC, el que luego la descifra y pone el resultado en el boleto generado. De este modo, Bob sabrá con toda seguridad que la clave de sesión está vinculada a la petición original de Alicia de hablar con él.

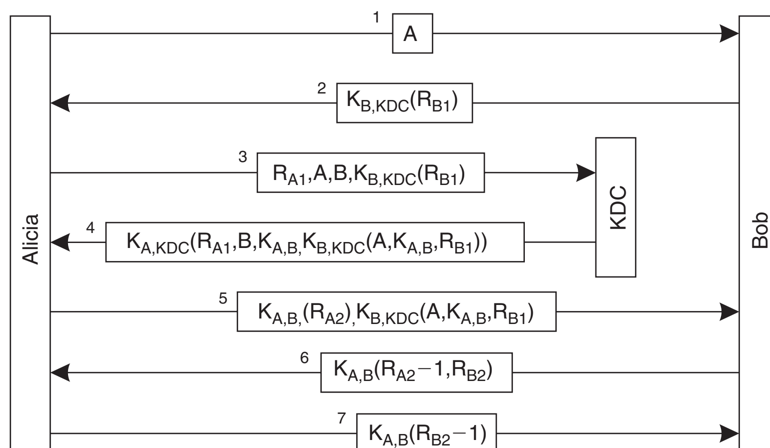


Figura 9-18. Protección contra la reutilización maliciosa de una clave de sesión previamente generada en el protocolo de Needham-Schroeder.

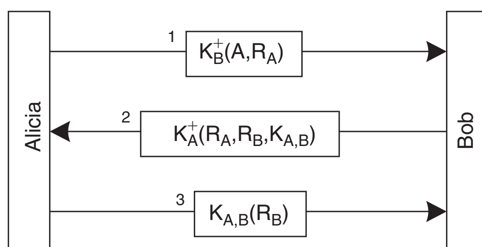


Figura 9-19. Autenticación mutua en un criptosistema de clave pública.

Autenticación con criptografía de clave pública

A continuación abordamos la autenticación con un criptosistema de clave pública que no requiere un KDC. De nueva cuenta, consideramos la situación en que Alicia desea establecer un canal seguro con Bob, y que ambos poseen la clave pública del otro. En la figura 9-19 se muestra un protocolo de autenticación típico basado en la criptografía de clave pública, el cual se explica a continuación.

Alicia comienza enviando un reto R_A a Bob cifrado con su clave pública K_B^+. La tarea de Bob es descifrar el mensaje y regresar el reto a Alicia. Como Bob es la única persona que puede descifrar el mensaje (usando la clave privada asociada con la clave pública que Alicia utilizó), Alicia sabrá que está hablando con Bob. Observemos la importancia de que Alicia esté segura de estar utilizando la clave pública de Bob y no la de alguien que está tratando de hacerse pasar por Bob. Cómo se pueden implementar tales garantías se analiza más adelante en este capítulo.

Cuando Bob recibe la petición de Alicia de establecer un canal, le regresa el reto descifrado junto con su propio reto R_B para autenticar a Alicia. Además, genera una clave de sesión $K_{A,B}$ que

pueda ser utilizada para establecer más comunicaciones. La respuesta de Bob al reto de Alicia, su propio reto, y la clave de sesión se incluyen en un mensaje cifrado con la clave pública K_A^+ que pertenece a Alicia, y mostrada como el mensaje 2 en la figura 9-19. Sólo Alicia será capaz de descifrar este mensaje con la clave pública privada K_A^- asociada con K_A^+.

Alicia, finalmente, regresa su respuesta al reto de Bob utilizando la clave de sesión $K_{A,B}$ generada por Bob. De esa manera, Alicia demuestra que sí podía descifrar el mensaje cifrado 2 y que, por tanto, sí es Alicia con la que en realidad está hablando Bob.

9.2.2 Integridad y confidencialidad del mensaje

Además de autenticación, un canal seguro también debe proporcionar garantías en cuanto a integridad y confidencialidad del mensaje. Integridad del mensaje significa que los mensajes están protegidos contra modificación subrepticia; la confidencialidad asegura que los mensajes no pueden ser interceptados y leídos por fisgones. La confidencialidad es fácil de establecer simplemente con cifrar un mensaje antes de enviarlo. El cifrado puede tener lugar o mediante una clave secreta compartida con el destinatario o, alternativamente, con la clave pública del destinatario. Sin embargo, la protección de un mensaje contra modificaciones es un poco más complicada, como se verá a continuación.

Firmas digitales

La integridad de un mensaje a menudo va más allá de la transferencia a través de un canal seguro. Consideremos la situación en que Bob acaba de vender a Alicia un disco fonográfico de colección en $500. La operación se realizó a través de correo electrónico. Al final, Alicia le envía un mensaje a Bob para confirmarle que comprará el disco en $500. Además de autenticación, existen por los menos dos cuestiones a tener en cuenta con respecto a la integridad del mensaje.

1. Alicia tiene que estar segura de que Bob no cambiará maliciosamente los $500 mencionados en su mensaje a una cifra mayor, y de que luego no diga que ella prometió más de $500.

2. Bob tiene que estar seguro de que Alicia no puede retractarse de que envió el mensaje, por ejemplo, porque cambió de idea.

Estas dos cuestiones pueden ser solventadas si Alicia firma digitalmente el mensaje, de tal forma que su firma quede vinculada de manera única a su contenido. La asociación única entre un mensaje y su firma evita que las modificaciones al mensaje pasen inadvertidas. Además, si la firma de Alicia puede ser verificada como genuina, luego ella no podrá decir que no firmó el mensaje.

Existen varias formas de colocar firmas digitales. Una muy popular es utilizar un criptosistema de clave pública tal como RSA, según muestra la figura 9-20. Cuando Alicia redacta un mensaje m para Bob, lo cifra con su clave *privada* K_A^- y se lo envía a Bob. Si también desea mantener en

secreto el contenido del mensaje, puede utilizar la clave pública de Bob y enviar $K_B^+(m,K_A^-(m))$, la cual combina *m* y la versión firmada por Alicia.

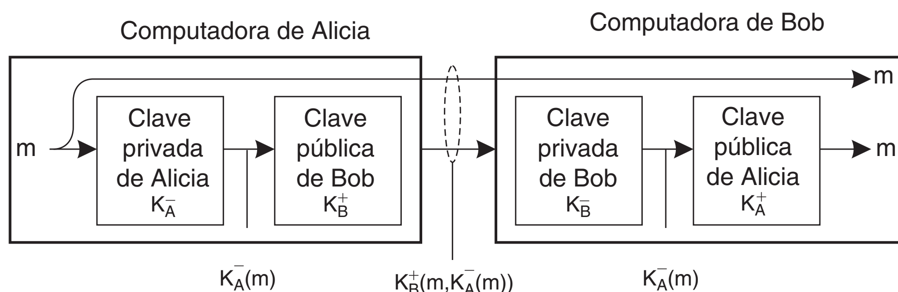


Figura 9-20. Firma digital de un mensaje empleando criptografía de clave pública.

Cuando Bob recibe el mensaje, lo descifra con la clave pública de Alicia. Si puede estar seguro de que la clave pública es realmente la de Alicia, el descifrado de la versión firmada de *m* y la comparación exitosa con *m* sólo pueden significar que provino de Alicia. Alicia está protegida contra cualesquiera modificaciones maliciosas por parte de Bob porque éste siempre tendrá que corroborar que la versión modificada de *m* fue también firmada por Alicia. Es decir, en esencia, el mensaje descifrado nunca cuenta como comprobación. También le interesa a Bob conservar la versión firmada de *m* para protegerse contra un rechazo por parte de Alicia.

Existen varios problemas asociados con este esquema, aunque el protocolo por sí mismo es correcto. En primer lugar, la validez de la firma de Alicia perdura sólo en tanto su clave privada permanezca secreta. Si Alicia desea deshacer el trato incluso después de enviar a Bob su confirmación, podría alegar que le robaron su clave privada antes de que enviara el mensaje.

Ocurre otro problema cuando Alicia decide cambiar su clave privada. Hacerlo no es una mala idea, ya que cambiar las claves de vez en cuando generalmente evita la intrusión. Sin embargo, una vez que Alicia cambia su clave, el mensaje enviado a Bob se vuelve inútil. Lo requerido en esos casos es una autoridad central que esté al tanto de cuándo se cambian las claves, además de utilizar marcas de tiempo al firmar los mensajes.

Otro problema con este esquema es que Alicia cifra todo el mensaje con su clave privada. Tal cifrado suele ser costoso en función de requerimientos de procesamiento (o incluso matemáticamente no factible ya que se asume que el mensaje interpretado como número binario está limitado por un máximo predefinido), y en realidad es innecesario. Recordemos que se debe asociar en forma única una firma con un solo mensaje específico. Un esquema más barato, y tal vez más elegante, es utilizar un resumen de mensaje.

Como ya se explicó, un resumen de mensaje es una cadena de bits de longitud fija *h* calculada a partir de un mensaje *m* de longitud arbitraria por medio de una función hash *H* criptográfica. Si *m* se cambia a m', su hash $H(m')$ será diferente de $h = H(m)$ de modo que una modificación sea fácil de detectar.

Para firmar digitalmente un mensaje, Alicia puede calcular primero un resumen de mensaje y posteriormente descifrarlo con su clave privada, como se muestra en la figura 9-21. El resumen cifrado se envía a Bob junto con el mensaje. Observemos que el mensaje se envía como texto común; todo mundo puede leerlo. Si se requiere confidencialidad, entonces el mensaje también deberá ser cifrado con la clave pública de Bob.

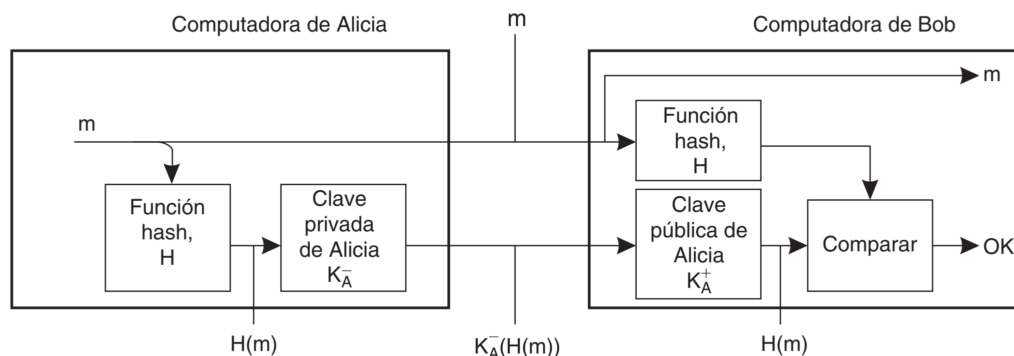


Figura 9-21. Firma digital de un mensaje empleando un resumen de mensaje.

Cuando Bob recibe el mensaje y su resumen cifrado, simplemente tiene que descifrar el resumen con la clave pública de Alicia y calcularlo por separado. Si el resumen calculado a partir del mensaje recibido y el resumen descifrado concuerdan, Bob puede estar seguro de que fue Alicia quien lo firmó.

Claves de sesión

Durante el establecimiento de un canal seguro, una vez completada la fase de autenticación, las personas que se están comunicando utilizan, por confidencialidad, una clave única de sesión compartida. La clave de sesión se desecha en forma segura cuando el canal ya no se utiliza. Una alternativa sería utilizar las mismas claves para confidencialidad que las utilizadas para establecer el canal seguro. No obstante, varios beneficios importantes se derivan del uso de claves de sesión (Kaufman y cols., 2003).

En primer lugar, cuando se utiliza una clave a menudo, es más fácil revelarla. En cierto sentido, las claves criptográficas se “deterioran” tal como las claves ordinarias. La idea básica es que si un intruso es capaz de interceptar una gran cantidad de datos cifrados con la misma clave, llega a ser posible implementar ataques para determinar ciertas características de las claves utilizadas y, posiblemente, revelar el texto común o la propia clave. Por eso resulta mucho más seguro utilizar las claves de autenticación lo menos que se pueda. Además, tales claves a menudo se intercambian utilizando algún mecanismo fuera de banda que requiere mucho tiempo, tal como el correo normal o el teléfono. Esa forma de intercambiar claves deberá mantenerse al mínimo.

Otra razón importante para generar una clave única para cada canal seguro es garantizar la protección contra ataques de reenvío como previamente ha sucedido en diversas ocasiones. Con el uso de una clave única cada vez que se establece un canal seguro, las personas que se están comunicando se encuentran protegidas por lo menos contra la reejecución de una sesión completa. Para protegerse contra el reenvío de mensajes individuales de una sesión previa, en general se requieren medidas adicionales tales como la inclusión de marcas de tiempo o de números secuenciales como parte del contenido del mensaje.

Supongamos que la integridad y confidencialidad del mensaje se lograron con la misma clave utilizada cuando se estableció la sesión. En ese caso, siempre que la clave se comprometa, un intruso puede ser capaz de descifrar mensajes transferidos durante una conversación pasada, claramente esto es algo no deseable. En cambio, es mucho más seguro utilizar claves por cada sesión ya que si la clave se compromete, en el peor de los casos, sólo una sesión se verá afectada. Los mensajes enviados durante otras sesiones permanecerán confidenciales.

Relacionado con este último punto está el hecho de que puede ser que Alicia desee intercambiar algunos datos confidenciales con Bob, pero no confía en él tanto como para darle información en la forma de datos cifrados con claves de larga duración. Es posible que desee reservar tales claves para mensajes altamente confidenciales que intercambia con personas en las que realmente confía. En esos casos, el uso de una clave de sesión relativamente barata para comunicarse con Bob es suficiente.

De manera general, las claves de autenticación a menudo se establecen de tal forma que su reemplazo resulte muy caro. Por consiguiente, la combinación de tales claves de larga duración con claves de sesión mucho más baratas y más temporales a menudo es una buena alternativa para implementar canales seguros para el intercambio de datos.

9.2.3 Comunicación segura de un grupo

Hasta ahora, la atención se ha concentrado en el establecimiento de un canal de comunicación seguro entre dos partes. En sistemas distribuidos, sin embargo, a menudo es necesario habilitar la comunicación segura entre más de dos partes. Un ejemplo típico es el de un servidor replicado donde toda la comunicación entre las réplicas deberá protegerse contra modificación, fabricación, e intercepción, como en el caso de canales seguros bipartitas. En esta sección, examinamos más a fondo la comunicación segura de un grupo.

Comunicación confidencial de un grupo

En primer lugar, consideramos el problema de proteger la comunicación entre un grupo de *N* usuarios contra el fisgoneo. Para garantizar la confidencialidad, un esquema simple es que todos los miembros del grupo compartan una clave secreta, la cual se utiliza para cifrar y descifrar todos los mensajes transmitidos entre los miembros del grupo. Como en este esquema todos los miembros comparten la clave secreta, es necesario confiar en que la conservarán secreta. Este único requisito previo propicia que el uso de una sola clave secreta compartida para la comunicación confidencial de un grupo sea más vulnerable a ataques en comparación con los canales seguros bipartitas.

Una solución alternativa es utilizar una clave secreta compartida diferente entre cada par de miembros del grupo. En cuanto un miembro comience a filtrar información los otros miembros simplemente pueden dejar de enviarle mensajes, pero continuar utilizando las claves secretas que utilizaban para comunicarse entre sí. Sin embargo, en lugar de tener que conservar una clave, ahora será necesario mantener $N(N - 1)/2$ claves, lo cual puede ser un problema difícil por sí mismo.

El uso de un criptosistema de clave pública puede mejorar las cosas. En ese caso, cada miembro dispone de su propio par (*clave pública*, *clave privada*), en el cual la clave pública puede ser utilizada por todos los miembros para enviar mensajes confidenciales. En este caso, se requiere un total de N pares de claves. Si un miembro deja de ser confiable, simplemente es eliminado del grupo sin tener que comprometer las demás claves.

Servidores replicados seguros

Ahora consideremos un problema completamente diferente: un cliente emite una petición a un grupo de servidores replicados. Éstos pueden haber sido replicados por razones de tolerancia o desempeño defectuosos, pero en todo caso, el cliente espera que la respuesta sea confiable. En otros términos, a pesar de que el grupo de servidores esté sujeto a fallas bizantinas como se vio en el capítulo anterior, un cliente espera que la respuesta regresada no haya sido sometida a un ataque de seguridad. Semejante ataque podría ocurrir si uno o más servidores fueron corrompidos por un intruso.

Una solución para proteger al cliente contra tales ataques es recopilar las respuestas de todos los servidores y autenticar cada una. Si existe cierta mayoría entre las respuestas de los servidores no corrompidos (es decir, autenticados), el cliente puede estar seguro de que también la respuesta es correcta. Desafortunadamente, este método revela la replicación de los servidores, por lo que se viola la transparencia de replicación.

Reiter y colaboradores (1994) proponen una solución para implementar un servidor replicado seguro en la que se mantiene la transparencia de replicación. La ventaja de su esquema es que, como los clientes desconocen las réplicas existentes, resulta mucho más fácil agregar o eliminar réplicas en forma segura. Más adelante regresaremos a la administración de grupos seguros cuando se aborde la administración de claves.

La esencia de los servidores replicados transparentes radica en lo que se denomina **compartimiento de un secreto**. Cuando múltiples usuarios (o procesos) comparten un secreto, ninguno de ellos conoce el secreto completo. En vez de eso, el secreto puede ser revelado sólo si se reúnen todos los usuarios. Tales esquemas pueden ser extremadamente útiles. Consideremos, por ejemplo, el lanzamiento de un proyectil nuclear. Tal acción requiere generalmente autorización de por lo menos dos personas. Cada una de estas personas guarda una clave privada que deberá utilizarse en combinación con la otra para lanzar un proyectil. El uso de una sola clave no lo hará.

En el caso de servidores seguros, replicados, lo que se busca es una solución mediante la cual, a lo sumo, k de los N servidores sean capaces de producir una respuesta incorrecta, y de estos k servidores, cuando mucho $c \leq k$ hayan sido realmente corrompidos por un intruso. Observemos que este requerimiento hace que el servicio sea tolerante a k fallas, tal como se vio

en el capítulo anterior. La diferencia radica en que un servicio corrompido ahora se clasifica como defectuoso.

Ahora consideremos la situación en la cual los servidores son activamente replicados. En otros términos, se envía una petición al mismo tiempo a todos los servidores que posteriormente es manejada por cada servidor. Cada servidor produce una respuesta que regresa al cliente. Para un grupo de servidores replicados con seguridad, se requiere que cada servidor acompañe su respuesta con una firma digital. Si r_i es la respuesta del servidor S_i, sea $md(r_i)$ el resumen de mensaje calculado por el servidor S_i. Este resumen se firma con la clave privada K_i^- del servidor S_i.

Supongamos que se desea proteger al cliente contra, cuando mucho, c servidores corruptos. En otros términos, el grupo de servidores deberá ser capaz de tolerar la corrupción por, cuando mucho, c servidores y seguir siendo capaz de producir una respuesta en la que el cliente pueda confiar. Si las firmas de los servidores individuales se pudieran combinar de tal forma que se requieran $c + 1$ firmas para construir una firma *válida* para la respuesta, eso resolvería el problema. Es decir, se permite que los servidores replicados generen una firma válida secreta con la propiedad de que c servidores corruptos solos no son suficientes para producir esa firma.

Como ejemplo, consideremos un grupo de cinco servidores replicados que deberán ser capaces de tolerar dos servidores corruptos y seguir produciendo una respuesta en la que un cliente pueda confiar. Cada servidor S_i envía su respuesta r_i al cliente, junto con su firma $sig(S_i,r_i) = K_i^-(md(r_i))$. Por consiguiente, el cliente al final habrá recibido cinco tripletas $<r_i, md(r_i), sig(S_i,r_i)>$ a partir de los cuales deberá derivar la respuesta correcta. Esta situación se muestra en la figura 9-22.

El cliente calcula también cada resumen $md(r_i)$. Si r_i es incorrecta, entonces normalmente se detecta calculando $K_i^+(K_i^-(md(r_i)))$. Sin embargo, este método ya no puede ser aplicado porque no se puede confiar en ningún servidor individual. En vez de eso, el cliente utiliza una función de descifrado especial públicamente conocida, D, la cual requiere un conjunto $V = \{sig(S, r), sig(S', r'), sig(S'', r'')\}$ de *tres* firmas como entrada y produce un resumen único como salida:

$$d_{salida} = D(V) = D(sig(S,r), sig(S',r'), sig(S'',r''))$$

Para conocer más detalles sobre D, vea Reiter (1994). Existen $5!/(3!2!) = 10$ posibles combinaciones de tres firmas que el cliente puede utilizar como entrada para D. Si una de estas combinaciones produce un resumen correcto $md(r_i)$ para alguna respuesta r_i, entonces el cliente puede considerar a r_i como correcta. En particular, puede estar seguro de que por los menos tres servidores honestos produjeron la respuesta.

Para mejorar la transparencia de replicación, Reiter y Birman hicieron posible que cada servidor S_i transmita un mensaje que contiene su respuesta r_i a los otros servidores, junto con la firma asociada $sig(S_i,r_i)$. Cuando un servidor ha recibido por lo menos $c + 1$ de tales mensajes, incluido su propio mensaje, intenta calcular una firma válida para una de las respuestas. Si esto resulta bien para, por ejemplo, la respuesta r y el conjunto V de $c + 1$ firmas, el servidor envía r y V como un solo mensaje al cliente. Éste puede verificar posteriormente la corrección de r confrontando su firma, es decir, si $md(r) = D(V)$.

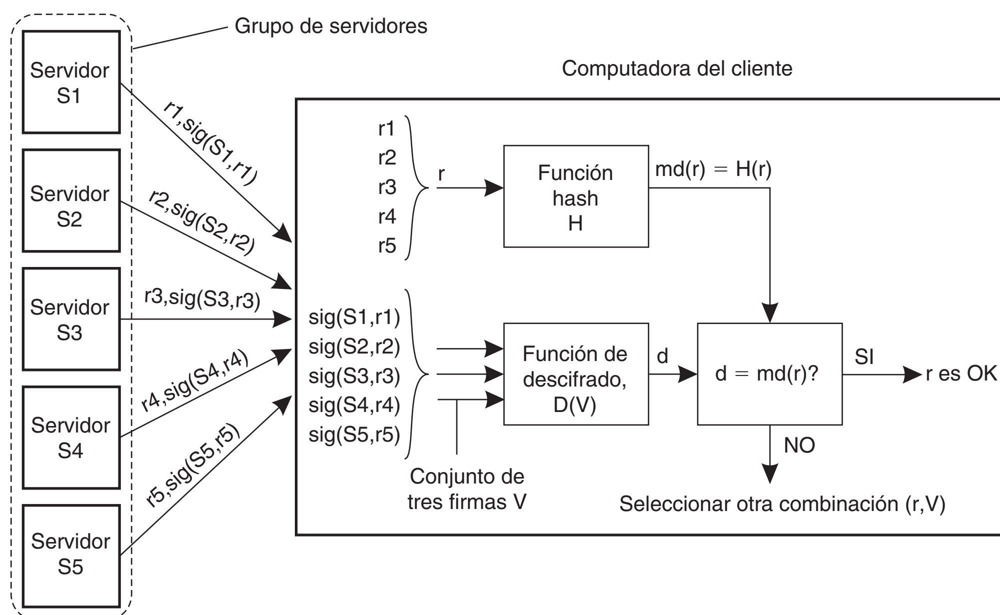


Figura 9-22. Compartimiento de una firma secreta en un grupo de servidores replicados.

Lo que se acaba de describir también se conoce como **esquema de umbral (m,n)** con, en este ejemplo, $m = c + 1$ y $n = N$, el número de servidores. En un esquema de umbral (m,n), un mensaje se divide en n partes, conocidas como **sombras** puesto que pueden utilizarse cualesquiera m sombras para reconstruir el mensaje original, pero con $m - 1$ o menos mensajes no se puede. Existen varias formas de construir esquemas de umbral (m,n). Los detalles se publicaron en Schneier (1996).

9.2.4 Ejemplo: Kerberos

Por ahora debe estar claro que la incorporación de seguridad en sistemas distribuidos no es algo trivial. Los problemas son provocados por el hecho de que todo el sistema debe ser seguro; si alguna parte no lo es, todo el sistema puede quedar comprometido. Para ayudar a construir sistemas distribuidos capaces de hacer cumplir una miríada de políticas de seguridad, se han desarrollado varios sistemas de apoyo que pueden ser utilizados como base para implementar un desarrollo posterior. Un sistema importante utilizado a menudo es el **Kerberos** (Steiner y cols., 1988; y Kohl y Neuman, 1994).

Kerberos se desarrolló en el MIT y está basado en el protocolo de autenticación de Needham-Schroeder antes descrito. En la actualidad existen dos versiones diferentes en uso, la versión 4 (V4) y la versión 5 (V5). Ambas son conceptualmente similares, aunque la V5 es mucho más flexible y

escalable. Una descripción detallada de V5 se halla en Neuman y colaboradores (2005), en tanto que Garman (2003) describe información práctica sobre la ejecución de Kerberos.

Kerberos puede ser visto como un sistema de seguridad que ayuda a los clientes a establecer un canal seguro con cualquier servidor que forme parte de un sistema distribuido. La seguridad está basada en claves secretas compartidas. Existen dos componentes distintos. El **AS** (*Authentication Server*; **servidor de autenticación**) es responsable de manejar una petición de inicio de sesión de un usuario. El AS autentifica a un usuario y proporciona una clave que puede ser utilizada para establecer canales seguros con los servidores. El establecimiento de canales seguros es manejado por un **TGS** (*Ticket Granting Service*; **servicio de otorgamiento de boletos**). El TGS entrega mensajes especiales, conocidos como **boletos**, que se utilizan para convencer a un servidor de que un cliente es realmente quien dice ser. A continuación presentamos ejemplos concretos de boletos.

Demos un vistazo a la forma en que Alicia inicia una sesión en un sistema distribuido que utiliza Kerberos y a cómo puede establecer un canal seguro con el servidor de Bob. Para iniciar una sesión en el sistema, Alicia puede utilizar cualquier estación de trabajo disponible. La estación de trabajo envía su nombre en texto común al AS, el cual regresa una clave de sesión $K_{A,TGS}$ y un boleto que Alicia deberá entregar al TGS.

El boleto regresado por el AS contiene la identidad de Alicia, junto con una clave secreta generada que Alicia y el TGS pueden utilizar para comunicarse entre sí. El boleto propiamente dicho será entregado al TGS por Alicia. Por consiguiente, es importante que nadie más excepto el TGS pueda leerlo. Es por esta razón que el boleto se cifra con la clave secreta $K_{AS,TGS}$ compartida entre el AS y el TGS.

Esta parte del procedimiento de inicio de sesión, se muestra como los mensajes 1, 2 y 3 en la figura 9-23. El mensaje 1 realmente no es un mensaje, sino que sirve para que Alicia escriba su nombre de inicio de sesión en una estación de trabajo. El mensaje 2 contiene ese nombre y es enviado al AS. El mensaje 3 contiene la clave de sesión $K_{A,TGS}$ y el boleto $K_{AS,TGS}(A,K_{A,TGS})$. Para garantizar la privacidad, el mensaje 3 se cifra con la clave secreta $K_{A,AS}$ compartida entre Alicia y el AS.

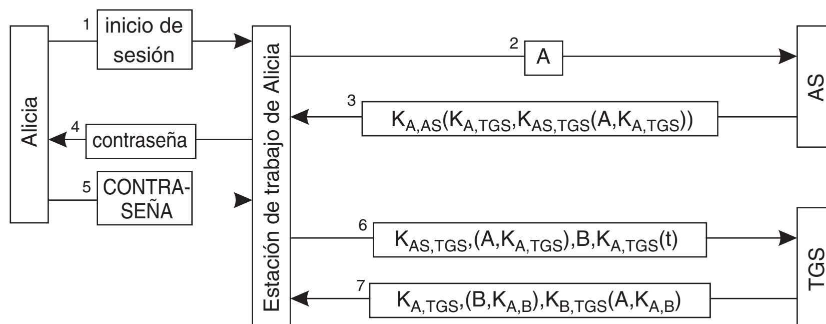


Figura 9-23. Autenticación en Kerberos.

Cuando la estación de trabajo recibe la respuesta del AS, le pide a Alicia su contraseña (mostrada como el mensaje 4), la cual utiliza para generar posteriormente la clave compartida $K_{A,AS}$.

(Es relativamente simple tomar una contraseña de cadena de caracteres, aplicar un hash criptográfico, y tomar luego los primeros 56 bits como la clave secreta). Observemos que este método no sólo tiene la ventaja de que la contraseña de Alicia nunca será enviada como texto común a través de la red, sino también que la estación de trabajo incluso no tenga que guardarla temporalmente. Además, en cuanto se genera la clave compartida $K_{A,AS}$, la estación de trabajo encuentra la clave de sesión $K_{A,TGS}$, y puede olvidarse de la contraseña de Alicia y utilizar solamente la clave secreta compartida $K_{A,AS}$.

Después de que se lleva a cabo esta parte de la autenticación, Alicia puede considerar que ya entró al sistema a través de la estación de trabajo actual. El boleto recibido del AS se guarda temporalmente (por lo general de 8 a 24 horas), y se utilizará para acceder a servicios remotos. Por supuesto, si Alicia abandona su estación de trabajo deberá destruir cualesquiera boletos guardados en la memoria caché. Si desea hablar con Bob, solicita al TGS que genere una clave de sesión para Bob, mostrada como el mensaje 6 en la figura 9-23. El hecho de que Alicia tenga en su poder el boleto $K_{AS,TGS}(A,K_{A,TGS})$ demuestra que ella es Alicia. El TGS responde con una clave de sesión $K_{A,B}$, de nueva cuenta encapsulada en un boleto que Alicia más tarde tendrá que pasarle a Bob.

El mensaje 6 también contiene una marca de tiempo, t, cifrada con la clave secreta compartida entre Alicia y el TGS. La marca de tiempo se utiliza para evitar que Chuck reenvíe maliciosamente el mensaje 6 y trate de establecer un canal con Bob. El TGS verificará la marca de tiempo antes de regresar un boleto a Alicia. Si difiere en más de unos cuantos minutos del tiempo actual, la petición de un boleto se rechaza.

Este esquema establece lo que se conoce como **inicio de sesión único**. En tanto Alicia no cambie de estación de trabajo, no hay necesidad de que se autentifique ante cualquier otro servidor que forme parte del sistema distribuido. Esta característica es importante cuando hay que habérselas con muchos servicios diferentes esparcidos a través de múltiples máquinas. En principio, los servidores han delegado en cierto modo la autenticación del cliente al AS y al TGS, y aceptarán peticiones de cualquier cliente que tenga un boleto válido. Desde luego, servicios tales como un inicio de sesión remoto requerirán que el usuario asociado tenga una cuenta, pero esto es independiente de la autenticación mediante Kerberos.

Establecer un canal seguro con Bob ahora resulta sencillo, y se muestra en la figura 9-24. Primero, Alicia envía un mensaje a Bob que contiene el boleto obtenido del TGS junto con una marca de tiempo cifrada. Cuando Bob descifra el boleto, se da cuenta de que Alicia está hablando con él porque sólo el TGS pudo haber estructurado el boleto. También encuentra la clave secreta $K_{A,B}$ que le permite verificar la marca de tiempo. En ese punto, Bob sabe que está hablando con Alicia y no con alguien más que maliciosamente reenvió el mensaje 1. Al responder con $K_{A,B}(t + 1)$, Bob le demuestra a Alicia que él sí es Bob.

9.3 CONTROL DE ACCESO

En el modelo cliente-servidor, el cual se ha utilizado hasta ahora, una vez que un cliente y un servidor establecen un canal seguro, el cliente puede emitir peticiones que deben ser ejecutadas por el servidor. Las peticiones implican realizar operaciones en recursos controlados por el servidor. Una

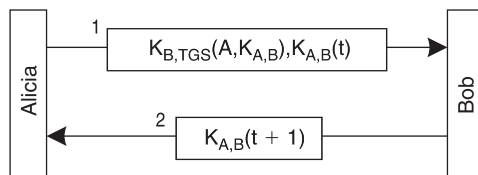


Figura 9-24. Establecimiento de un canal seguro en Kerberos.

situación general es la de un servidor de objetos que tiene varios objetos bajo su control. Una petición de un cliente implica, por lo general, invocar un método de un objeto específico. Tal petición puede ser realizada sólo si el cliente tiene **derechos de acceso** suficientes para conseguir dicha invocación.

De modo formal, la verificación de los derechos de acceso se conoce como **control de acceso**, mientras que **autorización** se refiere a la concesión de derechos de acceso. Los dos términos están fuertemente relacionados entre sí y con frecuencia se utilizan de modo intercambiable. Existen muchas formas de control de acceso. En primer lugar, abordamos algunos de los temas generales prestando atención especial a los diferentes modelos de manejo del control de acceso. Una manera importante de controlar el acceso a recursos es construir un cortafuego que proteja las aplicaciones o incluso toda una red. Los cortafuegos se estudian por separado. Con el advenimiento del código de movilidad, el control de acceso ya no podía realizarse únicamente con los métodos tradicionales. En vez de eso tuvieron que idearse técnicas nuevas, las cuales también se presentan en esta sección.

9.3.1 Temas generales de control de acceso

Para entender los diversos temas implicados en el control de acceso, en general se adopta el modelo simple mostrado en la figura 9-25. Este modelo consta de **sujetos** que emiten una petición de acceso a un **objeto**. Un objeto es algo muy parecido a los objetos que se han estado analizando hasta ahora. Se puede pensar en los objetos como encapsular su propio estado e implementar las operaciones en dicho estado. Las operaciones de un objeto que los sujetos pueden solicitar se realicen están disponibles mediante interfaces. Los sujetos pueden ser considerados mejor como procesos que actúan a nombre de los usuarios, aunque también pueden ser objetos que necesitan los servicios de otros objetos para realizar su propio trabajo.

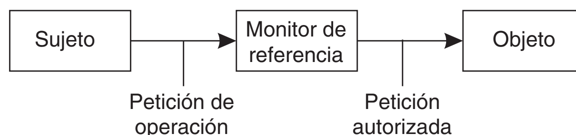


Figura 9-25. Modelo general de controlar el acceso a objetos.

Controlar el acceso a un objeto es todo lo que se refiere a protegerlo contra invocaciones realizadas por sujetos a los que no les está permitido tener algún método específico (o incluso ninguno) de los métodos realizados. También, la protección puede incluir temas de administración de objetos tales como crear, renombrar, o eliminar objetos. La protección a menudo se aplica mediante un programa llamado **monitor de referencia**. Un monitor de referencia registra qué sujeto puede hacer qué cosa, y decide si a un sujeto se le permite realizar una operación específica. A este monitor se le convoca (por ejemplo, mediante el sistema operativo confiable subyacente) cada vez que un objeto es invocado. Por consiguiente, es en extremo importante que el propio monitor de referencia sea a prueba de intrusiones: un atacante no debe ser capaz de inmiscuirse con él.

Matriz de control de acceso

Un método común usado para modelar los derechos de acceso de sujetos con respecto a objetos es construir una **matriz de control de acceso**. En esta matriz, cada sujeto está representado por una fila y cada objeto por una columna. Si la matriz se denota mediante M, entonces una entrada $M[s,o)]$ indica con precisión qué operaciones puede solicitar el sujeto s para que se realicen en el objeto o. En otros términos, siempre que un sujeto s solicite la invocación del método m del objeto o, el monitor de referencia deberá verificar si m aparece en $M[s,o)]$. Si m no aparece en $M[s,o)]$, la invocación falla.

Si consideramos que un sistema fácilmente puede necesitar dar soporte a miles de usuarios y que millones de objetos requieren protección, implementar una matriz de control de acceso como matriz verdadera no es la forma correcta de proceder. Muchas entradas de la matriz estarán vacías: en general, un solo sujeto tendrá acceso a relativamente pocos objetos. Por consiguiente, se siguen otras formas más eficientes para implementar una matriz de control de acceso.

Un método ampliamente aplicado es hacer que cada objeto mantenga una lista de derechos de acceso de sujetos que desean acceder a él. En esencia, esto significa que la matriz está distribuida en columnas a través de todos los objetos y que las entradas vacías se eliminan. Este tipo de implementación conduce a lo que se llama **ACL** (*Access Control List*; **lista de control de acceso**). Se supone que cada objeto tiene su propia ACL asociada.

Otro método es distribuir la matriz en filas dando a cada sujeto una lista de las **capacidades** que tiene para cada objeto. En otras palabras, una capacidad corresponde a una entrada en la matriz de control de acceso. No tener una capacidad para un objeto específico significa que el sujeto no tiene derechos de acceso para dicho objeto.

Una capacidad puede compararse a un boleto: su portador recibe ciertos derechos asociados con el boleto. También queda claro que un boleto deberá estar protegido contra modificaciones por parte de su portador. Un método particularmente adecuado en sistemas distribuidos, y el cual ha sido extensamente aplicado en Amoeba (Tanenbaum y cols., 1990), es proteger (una lista de) capacidades con una firma. Regresaremos a éstas y a otras materias más adelante, cuando analicemos la administración de seguridad.

La diferencia entre cómo se utilizan las ACL y las capacidades para proteger el acceso a un objeto se muestra en la figura 9-26. Utilizando las ACL, cuando un cliente envíe una petición a un servidor, el monitor de referencia del servidor verificará si conoce al cliente y si éste tiene permiso de realizar la operación solicitada, según ilustra la figura 9-26(a).

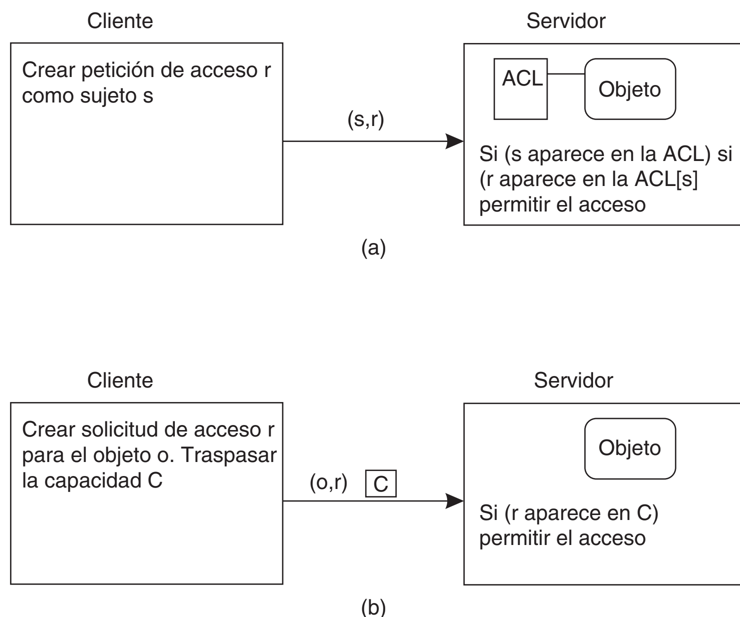


Figura 9-26. Comparación entre ACL y capacidades para proteger objetos. (a) Con una ACL. (b) Con capacidades.

Sin embargo, cuando utiliza capacidades, un cliente simplemente envía su solicitud (petición) al servidor. A éste no le interesa si conoce al cliente; la capacidad declara lo suficiente. Por tanto, el servidor necesita comprobar únicamente si la capacidad es válida y si la operación solicitada aparece en la capacidad. Este método de proteger objetos por medio de capacidades se muestra en la figura 9-26(b).

Dominios de protección

Las ACL y las capacidades ayudan a implementar con eficiencia una matriz de control de acceso al ignorar todas las entradas vacías. No obstante, una ACL o una lista de capacidades pueden llegar a ser lo bastante grandes si no se toman medidas adicionales.

Una forma general de reducir las ACL, es utilizar dominios de protección. Formalmente, un **dominio de protección** es un conjunto de pares (*objeto*, *derechos de acceso*). Para un objeto dado, cada par especifica con exactitud qué operaciones pueden realizarse (Saltzer y Schroeder, 1975). Las peticiones para realizar una operación siempre se emiten dentro de un dominio. Por consiguiente, siempre que un sujeto solicita realizar una operación en un objeto, el monitor de referencia busca primero el dominio de protección asociado con dicha petición. Acto seguido, dado el dominio, el monitor de referencia puede verificar si la petición tiene permiso de ser realizada. Existen diferentes usos de los dominios de protección.

Un método es construir **grupos** de usuarios. Consideremos, por ejemplo, una página web de la intranet de una compañía. Tal página deberá estar disponible para cada empleado, pero para nadie más. En lugar de agregar una entrada por cada empleado posible a la ACL de la página web, se puede decidir tener un grupo aparte *Employee* que contenga todos los empleados actuales. Siempre que un usuario acceda a la página web, el monitor de referencia sólo tiene que comprobar si el usuario es un empleado. Qué usuarios pertenecen al grupo *Employee* se mantiene en una lista aparte (la cual, desde luego, está protegida contra acceso no autorizado).

Con la introducción de grupos jerárquicos las cosas pueden hacerse más flexibles. Por ejemplo, si una organización tiene tres sucursales diferentes en, por ejemplo, Amsterdam, Nueva York, y San Francisco, es posible que desee subdividir su grupo *Employee* en subgrupos, uno por ciudad, que lleven a una organización como la mostrada en la figura 9-27.

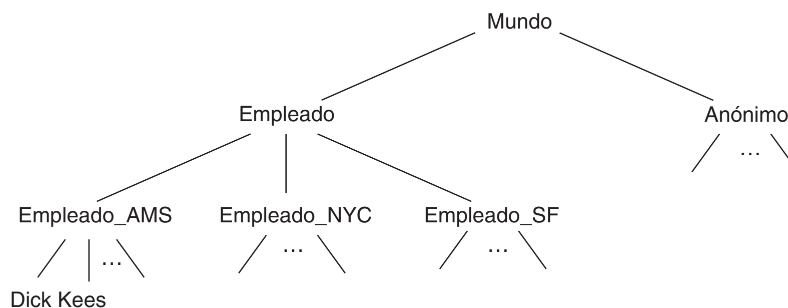


Figura 9-27. Organización jerárquica de dominios de protección como grupos de usuarios.

El acceso a las páginas web de la intranet de la organización deberá estar permitido a todos los empleados. Sin embargo, cambiar por ejemplo las páginas web asociadas con la sucursal de Amsterdam deberá estarle permitido sólo a un subconjunto de empleados basados en Amsterdam. Si el usuario Dick de Amsterdam desea leer una página web de la intranet, el monitor de referencia tiene que buscar primero en los subconjuntos *Employee_AMS*, *Employee_NYC*, y *Employee_SF* que juntos conforman el grupo *Employee*. Luego debe comprobar si uno de estos subconjuntos contiene a Dick. La ventaja de tener grupos jerárquicos es que la administración de la membresía para un grupo es relativamente fácil y que se pueden construir grupos muy grandes con eficiencia. Una desventaja evidente es que la búsqueda de un miembro puede resultar bastante costosa si la base de datos de membresías es distribuida.

En lugar de dejar que el monitor de referencia haga todo el trabajo, una alternativa es permitir que cada sujeto porte un **certificado** que contenga una lista de los grupos a los que pertenece. De esa manera, siempre que Dick desee leer una página de la intranet de la compañía, entregará su certificado al monitor de referencia para informarle que él es miembro del subgrupo *Employee_AMS*. Para garantizar que el certificado es genuino y que no ha sido manipulado, deberá estar protegido mediante, por ejemplo, una firma digital. Los certificados se consideran comparables a las capacidades. Más adelante regresaremos a estos temas.

En relación con el hecho de contar con dominios de protección, también es posible implementar los dominios de protección como **roles**. En un control de acceso basado en roles, un usuario siempre entra al sistema con un rol específico, el que a menudo está asociado con la función que el usuario cumple dentro de la organización (Sandhu y cols., 1996). Un usuario puede tener varias funciones. Por ejemplo, Dick podría ser al mismo tiempo jefe de departamento, gerente de proyecto, y miembro de un comité de búsqueda de personal. Según el rol que adopte cuando inicie una sesión, se le pueden asignar diferentes privilegios. En otros términos, su rol determina el dominio de protección (es decir, el grupo) en el cual operará.

Cuando se asignan roles a usuarios y se requiere que éstos adopten un rol específico al iniciar una sesión, también deberá serles posible que los cambien si es necesario. Por ejemplo, puede que se requiera permitir a Dick como jefe de departamento cambiar de vez en cuando a su rol de gerente de proyecto. Observemos que tales cambios son difíciles de expresar cuando se implementan dominios de protección sólo como grupos.

Además de utilizar dominios de protección, la eficiencia puede mejorarse aún más si los objetos se agrupan (jerárquicamente) con base en las operaciones que proporcionan. Por ejemplo, en lugar de considerar objetos individuales, éstos se agrupan de acuerdo con las interfaces que proporcionan, tal vez mediante una clasificación por subtipos [también conocida como herencia de interfaz, vea Gamma y cols., (1994)] para lograr cierta jerarquía. En este caso, cuando un sujeto solicita realizar una operación en un objeto, el monitor de referencia busca a qué interfaz pertenece la operación. Acto seguido verifica si el sujeto tiene permiso de invocar la operación que pertenece a dicha interfaz en lugar de si tiene permiso de invocar la operación para el objeto específico.

También es posible combinar los dominios de protección y agrupar los objetos. Utilizando ambas técnicas, junto con estructuras de datos específicas y operaciones restringidas en objetos, Gladney (1997) describe cómo implementar las ACL para conjuntos muy grandes de objetos utilizados en bibliotecas digitales.

9.3.2 Cortafuegos (Firewall)

Hasta ahora, se ha demostrado cómo se puede establecer la protección mediante técnicas criptográficas, combinadas con alguna implementación de una matriz de control de acceso. Estos métodos funcionan muy bien en tanto que todas las partes en comunicación actúen de acuerdo con el mismo conjunto de reglas. Tales reglas pueden ser aplicadas cuando se desarrolle un sistema distribuido autónomo aislado del resto del mundo. Sin embargo, las cosas se complican más cuando se permite el acceso a extraños a los recursos controlados por un sistema distribuido. Ejemplos de tales accesos incluyen envío de correo, descarga de archivos, carga de formas fiscales, etcétera.

Para proteger los recursos en estas circunstancias, se requiere un método diferente. En la práctica, sucede que el acceso externo a cualquier parte de un sistema distribuido es controlado por una clase especial de monitor de referencia conocido como **firewall** (Cheswick y Bellovin, 2000, y Zwicky y cols., 2000). En esencia, un cortafuego desconecta cualquier parte de un sistema distribuido del mundo externo, como se muestra en la figura 9-28. Todos los paquetes salientes, pero sobre todo los entrantes, se canalizan a través de una computadora especial y se inspeccionan antes

de permitirles el paso. El tráfico no autorizado se desecha y no se le permite continuar. Un tema importante es que el cortafuego propiamente dicho debe estar fuertemente protegido contra cualquier clase de amenaza de seguridad: nunca deberá fallar.

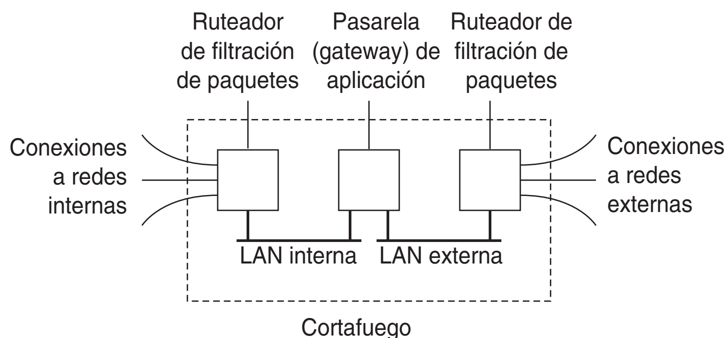


Figura 9-28. Implementación común de un cortafuego.

Los cortafuegos, en esencia, vienen en dos sabores diferentes que a menudo se combinan. Un tipo importante de cortafuego es una **compuerta de filtración de paquetes**. Este tipo de cortafuego opera como ruteador y toma decisiones en cuanto a dejar pasar o no un paquete de red con base en la dirección de origen o destino contenida en el encabezado del paquete. Típicamente, la compuerta de filtración de paquetes mostrada en la LAN externa de la figura 9-28 protege contra paquetes entrantes, en tanto que la LAN interna filtra los salientes.

Por ejemplo, para proteger un servidor web interno contra peticiones de hosts que no están en la red interna, una compuerta de filtración de paquetes podría decidir rechazar todos los paquetes entrantes dirigidos al servidor web.

Más sutil es la situación en que la red de una compañía consta de múltiples redes de área local conectadas, por ejemplo, mediante una red SMDS como se vio antes. Cada LAN puede estar protegida por medio de una compuerta de filtración de paquetes, configurada para que deje pasar el tráfico entrante sólo si proviene de un host localizado en una de las otras LAN. De este modo, se puede establecer una red virtual privada.

El otro tipo de cortafuego es una **compuerta a nivel de aplicación**. Por contraste con una compuerta de filtración de paquetes, la cual inspecciona sólo el encabezado de paquetes de red, este tipo de cortafuego en realidad inspecciona el contenido de un mensaje entrante o saliente. Un ejemplo típico es una compuerta de correo que desecha el correo entrante o saliente que excede cierto tamaño. Existen compuertas de correo más complejas que son, por ejemplo, capaces de filtrar correo electrónico basura.

Otro ejemplo de una compuerta a nivel de aplicación es una implementación que permite el acceso a un servidor de biblioteca externo, pero que proporciona sólo extractos de documentos. Si un usuario externo desea más, se inicia un protocolo de pago electrónico. Los usuarios localizados en el interior del cortafuego tienen acceso directo al servicio de biblioteca.

Una clase especial de compuerta a nivel de aplicación es la conocida como **compuerta proxy**. Este tipo de cortafuego funciona como etapa frontal ante una clase específica de aplicación, y garantiza que pasarán sólo aquellos mensajes que satisfagan ciertos criterios. Consideremos, por ejemplo, navegar por la web. Tal como analizamos en la siguiente sección, muchas páginas web contienen scripts o applets que deben ser ejecutados en el buscador de un usuario. Para evitar que semejante código se descargue hacia la LAN interna, todo el tráfico web podría ser encauzado a través de una compuerta proxy. La compuerta acepta peticiones http regulares, o desde adentro o de afuera del cortafuego. En otros términos, aparece ante sus usuarios como un servidor web normal. No obstante, filtra todo el tráfico saliente o entrante, ya sea desechando ciertas peticiones y páginas o modificándolas cuando contienen un código ejecutable.

9.3.3 Código móvil seguro

Tal como estudiamos en el capítulo 3, un desarrollo importante en sistemas distribuidos modernos es la capacidad de migrar el código entre servidores en lugar de sólo migrar datos pasivos. Sin embargo, el código móvil introduce varias amenazas de seguridad serias. Entre otras cosas, cuando se envía un agente a través de internet, su propietario desea protegerse contra servidores malévolos que traten de robarse o modificar la información transportada por el agente.

Otra cuestión importante es que los servidores necesitan estar protegidos contra agentes maliciosos. La mayoría de los usuarios de sistemas distribuidos no son expertos en tecnología de sistemas y no tienen forma de saber si el programa que están descargando de otro servidor no corromperá su computadora. En muchos casos puede ser difícil incluso para un experto detectar si realmente un programa se ha descargado por completo.

A menos que se tomen medidas de seguridad una vez que un sistema malicioso se ha establecido en una computadora, es fácil que corrompa su servidor. Se enfrenta un problema de control de acceso: no se deberá permitir el acceso no autorizado a los recursos del servidor. Como se verá, proteger un servidor contra programas maliciosos descargados no siempre es fácil. El problema no se reduce a evitar la descarga de programas. En su lugar, lo que se busca es un código móvil de soporte que pueda permitir el acceso a los recursos locales de una manera flexible aunque totalmente controlada.

Protección de un agente

Antes de considerar la protección de un sistema de computadora contra un código malicioso descargado, primero veamos la situación opuesta. Consideremos un agente móvil que transita por un sistema distribuido a nombre de un usuario. Tal agente puede estar buscando el boleto de avión más barato de Nairobi a Malindi y ha sido autorizado por su propietario para que haga una reservación en cuanto encuentre un vuelo. Con este propósito, el agente puede portar una tarjeta de crédito electrónica.

Desde luego, en este caso se requiere protección. Siempre que el agente entre a un servidor, éste no tendrá permiso de robar la información de la tarjeta de crédito del agente. También, el

agente deberá estar protegido contra modificaciones que hagan que el propietario pague mucho más de lo que en realidad se requiere. Por ejemplo, si la agencia Chuck's Cheaper Charters puede ver que el agente aún no ha visitado a su competidor más barato Alicia Airlines, Chuck podría prevenir a partir de un cambio de agente que se visite el servidor de Alicia Airlines. Otros ejemplos que requieren protección de un agente contra ataques de un servidor enemigo incluyen la destrucción maliciosa de un agente o la manipulación de un agente de modo que ataque o robe a su propietario.

Por desgracia, la protección total de un agente contra toda clase de ataques es imposible (Farmer y cols., 1996). Esta imposibilidad es provocada principalmente por el hecho de que no se pueden dar garantías rigurosas de que un servidor hará lo que promete. Un método alternativo es, en consecuencia, organizar agentes de modo que por lo menos puedan ser detectadas las modificaciones. El sistema Ajanta (Karnik y Tripathi, 2001) sigue este método. Ajanta proporciona tres mecanismos que permiten al propietario de un agente detectar que éste ha sido manipulado: estado de sólo lectura, registros de sólo adición, y revelación selectiva de estado a ciertos servidores.

El **estado de sólo lectura** de un agente Ajanta consta de una colección de elementos de datos firmada por el propietario del agente. La firma tiene lugar cuando se construye e inicializa el agente antes de ser enviado a otro servidor. El propietario construye primero un resumen de mensaje que posteriormente cifra con su clave privada. Cuando el agente llega a un servidor, éste puede detectar con facilidad si el estado de sólo lectura ha sido manipulado al verificar el estado contra el resumen de mensaje del estado original firmado.

Para permitir que un agente reúna información mientras se desplaza entre servidores, Ajanta proporciona **registros de sólo adición**. Estos registros se caracterizan por el hecho de que los datos sólo pueden ser anexados al registro; no hay forma de que los datos puedan ser eliminados o modificados sin que el propietario sea capaz de detectarlo. El uso de un registro de sólo adición funciona como sigue. Inicialmente, el registro está vacío y tiene sólo una suma de verificación asociada C_{inic} calculada como $C_{inic} = K^{+}_{propietario}(N)$, donde $K^{+}_{propietario}$ es la clave pública del propietario del agente, y N es un nonce secreto conocido sólo por el propietario.

Cuando el agente llega a un servidor que desea entregarle algunos datos X, S anexa X al registro luego firma X con su firma $sig(S,X)$, y calcula una suma de verificación.

$$C_{nueva} = K^{+}_{propietario}(C_{vieja},\ sig(S,X),\ S)$$

donde C_{vieja} es la suma de verificación previamente utilizada.

Cuando el agente regresa a su propietario, éste puede verificar fácilmente si el registro ha sido manipulado. El propietario comienza leyendo el registro por el final al calcular sucesivamente $K^{-}_{propietario}(C)$ en la suma de verificación C. Cada iteración regresa una suma de verificación $C_{siguiente}$ para la siguiente iteración, junto con $sig(S,X)$ y S para algún servidor S. El propietario puede verificar luego si el entonces último elemento del registro concuerda o no con $sig(S,X)$. De ser así, el elemento es extraído y procesado, tras de lo cual se toma el siguiente paso de iteración. La iteración se detiene al llegar a la suma de verificación inicial o cuando el propietario advierte que el registro ha sido manipulado porque la firma no concuerda.

Por último, Ajanta da soporte a la **revelación selectiva** de estado al proporcionar una matriz de elementos de datos, donde cada entrada está pensada para un servidor designado. Cada entrada se cifra con la clave pública del servidor designado para garantizar la confidencialidad. Toda la matriz es firmada por el propietario del agente para garantizar la integridad en su conjunto. En otros términos, si *cualquier* entrada es modificada por un servidor malicioso, cualesquiera de los servidores designados lo advertirá y emprenderá la acción apropiada.

Además de proteger a un agente contra servidores maliciosos, Ajanta proporciona varios mecanismos para proteger servidores contra agentes maliciosos. Como se verá a continuación, muchos de estos mecanismos también son proporcionados por otros sistemas que soportan el código móvil. Más información sobre Ajanta se encuentra en Tripathi y colaboradores (1999).

Protección del destino

Aun cuando el código móvil de protección contra un servidor malicioso es importante, se ha prestado más atención a la protección de servidores contra código móvil malicioso. Si el envío de un agente al mundo exterior se considera demasiado peligroso, generalmente un usuario tendrá alternativas para realizar el trabajo para el cual estaba pensado el agente. Sin embargo, a menudo no existen alternativas en cuanto a dejar que un agente entre a su sistema, aparte de bloquearlo por completo. Por consiguiente, si se decide que el agente puede entrar, el usuario necesita implementar un control completo sobre lo que el agente puede hacer.

Como se acaba de ver, aunque proteger un agente contra modificaciones puede ser imposible, por lo menos es posible que el propietario del agente detecte que se hicieron modificaciones. En el peor de los casos, el propietario tendrá que rechazar al agente cuando regresa, pero de otro modo ningún daño habrá sido hecho. No obstante, cuando hay que habérselas con agentes maliciosos, no sirve de nada detectar que los recursos han sido hostigados. En su lugar, es esencial proteger todos los recursos contra acceso no autorizado por medio de un código descargado.

Una forma de protección es construir una caja de arena. Una **caja de arena** es una técnica mediante la cual un programa descargado se ejecuta de modo que cada una de sus instrucciones pueda ser controlada a cabalidad. Si se intenta ejecutar una instrucción prohibida por el servidor, tal ejecución del programa se detendrá. Asimismo, la ejecución se detiene cuando una instrucción accede a ciertos registros o áreas de la memoria sin que el servidor lo haya permitido.

La implementación de una caja de arena no es fácil. Una forma de abordar el problema es verificar el código ejecutable cuando se descarga e insertarle instrucciones adicionales para situaciones que puedan ser comprobadas sólo en tiempo de ejecución (Wahbe y cols., 1993). Afortunadamente, las cosas se simplifican cuando se trata de código interpretado. A continuación analizaremos brevemente el método aplicado en Java [vea también (MacGregor y cols., (1998)]. Cada programa Java se compone de varias clases a partir de las cuales se crean los objetos. No existen variables ni funciones globales; todo tiene que ser declarado como parte de una clase. La ejecución del programa inicia con un método llamado `main`. Un programa Java se compila en la forma de un conjunto de instrucciones interpretadas por lo que se conoce como **Java Virtual Machine (JVM)**. Para que un cliente descargue y ejecute un programa Java compilado es necesario,

por consiguiente, que el proceso del cliente haga funcionar la JVM. Ésta se encargará posteriormente de ejecutar el programa descargado al interpretar cada una de sus instrucciones, comenzando con las instrucciones que conforman main.

En una caja de arena Java, la protección se inicia asegurándose de que se puede confiar en el componente que maneja la transferencia de un programa a la máquina cliente. Un conjunto de **cargadores de clase** se encarga de la descarga en Java. Cada cargador de clase es responsable de extraer una clase específica de un servidor e instalarla en el espacio de la dirección del cliente de modo que la JVM pueda crear objetos con ella. Como un cargador de clase es simplemente otra clase Java, es posible que un programa descargado contenga sus propios cargadores de clase. Lo primero que hace una caja de arena es ver que se utilicen exclusivamente cargadores de clase confiables. En particular, no se permite que un programa Java cree sus propios cargadores de clase mediante los cuales podría evadir la forma en que normalmente se maneja la carga de clase.

El segundo componente de una caja de arena Java consta de un **verificador de código de bytes**, el cual comprueba si una clase descargada obedece las reglas de seguridad de la caja de arena. En particular, verificar comprueba que la clase no contenga ningunas instrucciones ilegales o instrucciones que pudieran corromper de algún modo la pila o la memoria. No todas las clases son verificadas, como se muestra en la figura 9-29; sólo aquellas que se descargan de un servidor externo al cliente. En general, se confía en las clases localizadas en la máquina del cliente, aun cuando su integridad también podría ser fácil de comprobar.

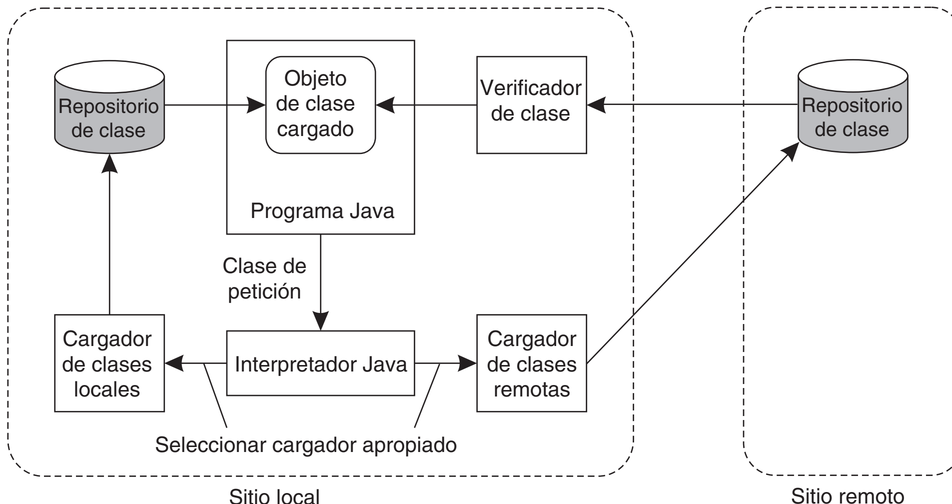


Figura 9-29. Organización de la caja de arena Java.

Por último, cuando una clase ha sido descargada y verificada con seguridad, la JVM puede producir objetos a partir de ella y ejecutar los métodos de éstos. Para impedir aún más que los objetos accedan sin autorización a los recursos del cliente, se utiliza un **administrador de seguridad** para realizar varias comprobaciones en tiempo de ejecución. Se obliga a que los programas

Java sean pensados para descargarse utilizando un gerente de seguridad; no hay forma de que puedan evadirlo. Esto significa, por ejemplo, que cualquier operación de E/S es examinada en cuanto a validez y no será realizada si el gerente de seguridad dice “no”. El gerente de seguridad desempeña, por tanto, el rol de un monitor de referencia; el cual ya se estudió con anterioridad.

Un gerente de seguridad típico denegará la realización de muchas operaciones. Por ejemplo, virtualmente todos los gerentes de seguridad prohiben el acceso a archivos locales y permiten que un programa sólo establezca comunicación con el servidor del que provino. Desde luego, tampoco se permite manipular la JVM. No obstante, se permite que un programa acceda a la biblioteca de gráficos con propósitos de visualización y para captar eventos tales como mover el ratón u oprimir sus botones.

El gerente de seguridad Java original implementó una política de seguridad bastante estricta en la cual no hizo ninguna distinción entre los diferentes programas descargados, ni siquiera entre programas de diferentes servidores. En muchos casos, el modelo de caja de arena Java inicial estaba demasiado restringido y se requería más flexibilidad. A continuación, presentamos un método alternativo que actualmente se sigue.

Un método en armonía con el uso de la caja de arena, pero que ofrece un poco de más flexibilidad, es crear un campo de juego para un código móvil descargado (Malkhi y Reiter, 2000). Un **campo de juego** es una máquina designada distinta y reservada exclusivamente para ejecutar un código móvil. Los recursos locales con respecto al campo de juego, tales como archivos o conexiones de red a servidores externos, están disponibles para programas que se ejecuten en el campo de juego y quedan sujetos a los mecanismos de protección normales. Sin embargo, los recursos locales con respecto a otras máquinas están físicamente desconectados del campo de juego y no pueden ser accesados por códigos descargados. Los usuarios de estas otras máquinas pueden acceder al campo de juego en la forma tradicional, por ejemplo, usando RPC. No obstante, nunca se descarga un código móvil a máquinas que no se encuentren en el campo de juego. Esta distinción entre una caja de arena y un campo de juego se muestra en la figura 9-30.

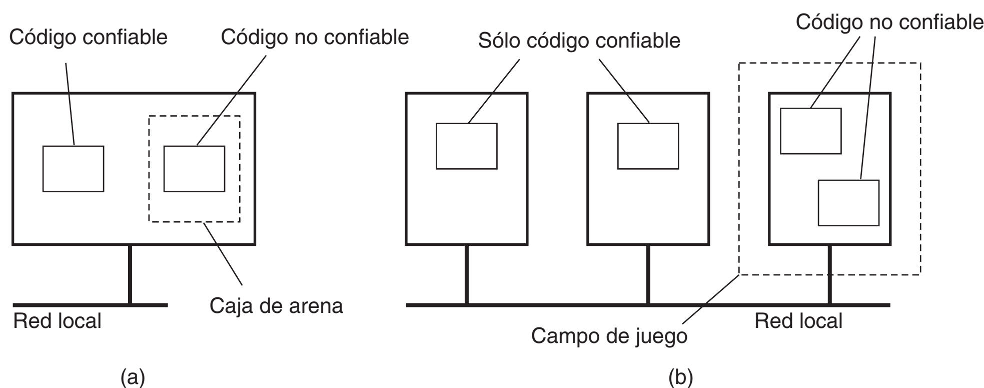


Figura 9-30. (a) Caja de arena. (b) Campo de juego.

Un siguiente paso para incrementar la flexibilidad, es requerir que cada programa descargado pueda ser autenticado, y para hacer que posteriormente se cumpla una política de seguridad especí-

fica basada en la procedencia del programa. Demandar que los programas puedan ser autenticados es relativamente fácil: el código móvil puede ser firmado como cualquier otro documento. Este método de **firma de código** a menudo se aplica como alternativa de la caja de arena. En realidad, sólo se aceptan códigos de servidores confiables.

Sin embargo, la parte difícil es hacer que se cumpla una política de seguridad. Wallach y colaboradores (1997) proponen tres mecanismos para el caso de programas Java. El primer mecanismo está basado en el uso de referencias de objeto como capacidades. Para acceder a un recurso local, digamos un archivo, un programa debe haber recibido una referencia a un objeto específico que maneja operaciones de archivo cuando se descarga. Si no se da ninguna referencia, no hay forma de que los archivos puedan ser accesados. Este principio se muestra en la figura 9-31.

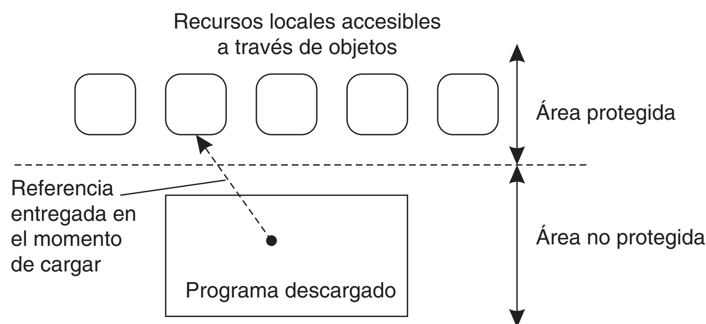


Figura 9-31. El principio de la utilización de referencias a objetos Java como capacidades.

Todas las interfaces para objetos que implementan el sistema de archivos se encuentran inicialmente ocultas al programa simplemente con no manejar ninguna referencia a estas interfaces. La demandante valoración de Java garantiza que sea imposible construir una referencia a una de estas interfaces en tiempo de ejecución. Además, se puede utilizar la propiedad de Java para mantener ciertas variables y métodos completamente internos para una clase. En particular, se puede evitar que un programa produzca sus propios objetos de manejo de archivos, si se oculta esencialmente la operación que crea objetos nuevos a partir de una clase dada. (En terminología Java, un constructor es hecho privado para su clase asociada.)

El segundo mecanismo para aplicar una política de seguridad es una **introspección de pila (extendida)**. En esencia, cualquier invocación a un método *m* de un recurso local está precedida por una llamada a un procedimiento especial enable_privilege que comprueba si quien llama está autorizado para invocar *m* en relación con ese recurso. Si la invocación es autorizada, quien llama recibe privilegios temporales durante el periodo de llamada. Antes de regresar el control al invocador cuando se termina *m*, el procedimiento especial disable_privilege es invocado para deshabilitar los privilegios.

Para forzar las llamadas hacia enable_privilege y disable_privilege, se podría requerir que un desarrollador de interfaces para recursos locales las inserte en los lugares apropiados. Sin embargo, es mucho mejor dejar que el interpretador Java maneje las llamadas automáticamente.

Éste es el método estándar seguido en, por ejemplo, un navegador para manejar los applets Java. Una solución elegante es la siguiente. Siempre que se invoca un recurso local, el interpretador Java invoca automáticamente el procedimiento enable_privilege, el cual comprueba en seguida si la invocación está permitida. De ser así, se coloca una invocación disable_privilege en la pila para garantizar que se deshabiliten los privilegios cuando la invocación al método regrese. Este enfoque impide que programadores maliciosos contravengan las reglas.

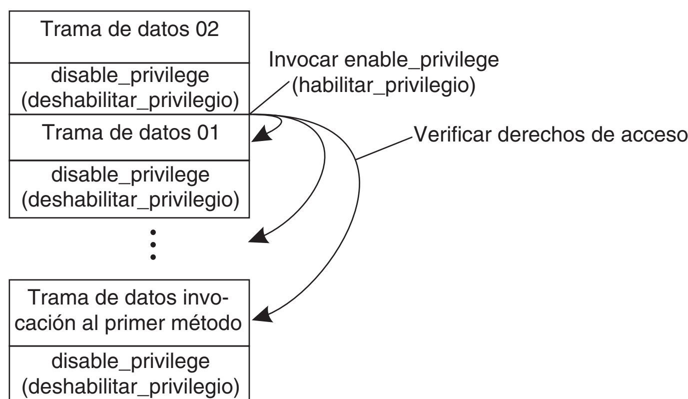


Figura 9-32. El principio de introspección de pila o trama de datos.

Otra ventaja importante de utilizar la pila es que hace posible una mejor forma de comprobar los privilegios. Supongamos que un programa invoca un objeto local *O*1, el cual, a su vez, invoca el objeto *O*2. Aunque *O*1 puede tener permiso de invocar *O*2, si no se confía en que el invocador de *O*1 invoque un método específico perteneciente a *O*2, entonces no se permitirá esta cadena de invocaciones. La introspección de pila facilita la comprobación de tales cadenas, ya que el interpretador simplemente necesita inspeccionar cada marco de pila comenzando por la parte superior para ver si existe uno que tenga los privilegios correctos habilitados (en cuyo caso se autoriza la llamada), o si existe un marco que explícitamente prohiba el acceso al recurso actual (en cuyo caso la invocación se termina de inmediato). Este método se muestra en la figura 9-32.

En esencia, la introspección de pila permite anexar privilegios a clases o métodos y comprobarlos para cada invocador por separado. De este modo, es posible implementar dominios de protección basados en clases, tal como se explica con todo detalle en Gong y Schemers (1998).

El tercer método para aplicar una política de seguridad es por medio de la **administración del espacio de nombre**. La idea se explica a continuación. Para permitir el acceso de programas a recursos locales, primero necesitan obtener el acceso con la inclusión de los archivos apropiados que contienen las clases que implementan esos recursos. La inclusión requiere que se dé un nombre al interpretador, quien luego lo reduce a una clase, la cual se carga posteriormente en tiempo de ejecución. Para hacer que un programa descargado específico cumpla una política de seguridad, el mismo nombre puede ser descompuesto en diferentes clases, según de dónde provenga el programa descargado. Típicamente, los cargadores de clase se encargan de la descomposición del nombre, y

necesitan ser adaptados para implementar este método. En Wallach y colaboradores (1997) se refiere cómo hacer esto.

El método descrito hasta ahora asocia privilegios con clases o métodos con base en el origen del programa. Por medio del interpretador Java, es posible aplicar políticas de seguridad empleando los mecanismos ya descritos. En este sentido, la arquitectura de seguridad llega a ser altamente dependiente del lenguaje y no tendrá que ser diseñada de nuevo para otros lenguajes. Soluciones independientes del lenguaje, como por ejemplo las descritas en Jaeger y colaboradores (1999), requieren un método más general sobre cómo aplicar la seguridad, aunque también son más difíciles de implementar. En estos casos, se requiere el soporte de un sistema operativo seguro que conozca el código móvil descargado y obligue a que todas las invocaciones a recursos locales se ejecuten a través del kernel donde posteriormente se verifiquen.

9.3.4 Negación de servicio

El control de acceso se refiere en general a garantizar en forma prudente que los recursos sean accesados sólo por procesos autorizados. Un tipo de ataque particularmente molesto relacionado con el control de acceso es evitar de modo malicioso que procesos autorizados accedan a recursos. Defensas contra tales **ataques de negación de servicio** (**DoS**, del inglés *denial of service*) se vuelven cada vez más importantes ya que los sistemas distribuidos se encuentran abiertos en internet. Cuando los ataques DoS provienen de una o varias fuentes, a menudo pueden ser manejados con efectividad. Las cosas llegan a ser más difíciles cuando se trata de una **negación de servicio distribuido** (**DDoS**, del inglés *distributed denial of service*).

En ataques DDoS, una enorme colección de procesos intenta anular conjuntamente un servicio distribuido en red. En estos casos, los atacantes a menudo tienen éxito al secuestrar un gran grupo de máquinas que sin saber participan en el ataque. Specht y Lee (2004) distinguen dos tipos de ataque; aquellos encaminados al agotamiento del ancho de banda y los encaminados al agotamiento de recursos.

El agotamiento del ancho de banda se logra simplemente con enviar muchos mensajes a una sola máquina. El efecto es que los mensajes normales difícilmente llegarán al receptor. Los ataques de agotamiento de recursos se concentran en permitir que el receptor agote los recursos en mensajes inútiles. Un ataque muy conocido de agotamiento de recursos es la inundación de TCP SYN. En este caso, el atacante intenta iniciar una enorme cantidad de conexiones (es decir, envía paquetes SYN como parte de un protocolo de diálogo tridireccional), aunque nunca responderá a reconocimientos del receptor.

No existe ningún método de protegerse contra ataques DDoS. Un problema es que los atacantes se aprovechan de víctimas inocentes al instalar en secreto software en sus máquinas. En estos casos, la única solución es hacer que las máquinas monitoreen su estado verificando sus archivos en cuanto a contaminación. Considerando la facilidad con que un virus puede esparcirse por internet, no es factible confiar sólo en esta medida preventiva.

Es mucho mejor monitorear continuamente el tráfico en la red, por ejemplo, comenzando por los ruteadores de salida por donde salen los paquetes de la red de una organización. La experiencia muestra que al rechazar los paquetes cuya dirección de procedencia no pertenece a la red de la

organización se pueden evitar muchos estragos. En general, mientras más paquetes puedan ser filtrados cerca de su origen, mucho mejor.

De manera alternativa, también es posible concentrarse en los ruteadores de entrada, es decir, donde el tráfico entra a la red de una organización. El problema es que la detección de un ataque en un ruteador de entrada es demasiado tardía puesto que la red probablemente ya no estará al alcance del tráfico regular. Es mejor hacer que ruteadores más distantes en internet, tales como en las redes de ISP, comiencen a rechazar paquetes cuando sospechen que un ataque está en proceso. Gil y Poletto (2001) siguen este método, donde un ruteador rechazará paquetes cuando advierta que la proporción entre el número de paquetes *hacia* un nodo específico es desproporcionado con respecto al número de paquetes *que salen* de dicho nodo.

En general, tiene que desplegarse una miríada de técnicas en tanto nuevos ataques continúen surgiendo. Un compendio práctico de la técnica más avanzada de negación y soluciones de ataques de servicio se encuentra en Mirkovic y colaboradores (2005); en Mirkovic y Reiher (2004) se presenta una taxonomía detallada.

9.4 ADMINISTRACIÓN DE LA SEGURIDAD

Hasta ahora se han considerado canales seguros y el control de acceso, pero apenas se ha tocado el tema de cómo, por ejemplo, se obtienen claves. En esta sección, examinamos más a fondo la administración de la seguridad. En particular, se distinguen tres temas diferentes. Primero, se tiene que considerar la administración general de claves criptográficas, y en especial la forma en que se distribuyen las claves públicas. Como resulta ser, los certificados desempeñan un rol importante aquí.

En segundo lugar, abordamos el problema de administrar con seguridad un grupo de servidores concentrándonos en el problema de agregar un nuevo miembro confiable para los miembros actuales. Claramente, de cara a los servicios distribuidos y replicados, es importante que no se comprometa la seguridad admitiendo un proceso malicioso para un grupo.

En tercer lugar, consideramos la administración de la autorización atendiendo las capacidades y lo que se conoce como certificados de atributo. Un tema importante en sistemas distribuidos, con respecto a la administración de la autorización, es que un proceso puede delegar algunos o todos sus derechos de acceso a otro proceso. La delegación de derechos en forma segura tiene sus propias sutilezas, tal como también se analiza en esta sección.

9.4.1 Administración de claves

Hasta aquí, hemos descrito varios protocolos criptográficos en los cuales se asumió (implícitamente) que estaban disponibles varias claves. Por ejemplo, en el caso de criptosistemas de clave pública, se supuso que el remitente de un mensaje tenía a su disposición la clave del destinatario, de tal suerte que podía cifrar el mensaje para asegurar su confidencialidad. Asimismo, en el caso de autenticación por medio de un centro de distribución de claves (KDC), supusimos que cada una de las partes ya compartía una clave secreta con el KDC.

Sin embargo, el establecimiento y la distribución de claves no es un asunto trivial. Por ejemplo, la distribución de claves secretas por medio de un canal inseguro es impensable y en muchos casos se tiene que recurrir a métodos que no están disponibles. También, se requieren mecanismos para revocar las claves, es decir, para impedir que una clave sea utilizada una vez que ha sido comprometida o invalidada. Por ejemplo, la revocación es necesaria cuando una clave ha sido comprometida.

Establecimiento de claves

Primero consideremos cómo se pueden establecer claves de sesión. Cuando Alicia desea establecer un canal seguro con Bob, puede utilizar primero la clave pública de Bob para iniciar la comunicación como se muestra en la figura 9-19. Si Bob acepta, posteriormente puede generar la clave de sesión y regresársela a Alicia cifrada con la clave pública de Alicia. Cifrando la clave de sesión compartida antes de su transmisión, puede ser transferida con seguridad a través de la red.

Se puede utilizar un esquema similar para generar y distribuir una clave de sesión cuando Alicia y Bob ya comparten una clave secreta. Sin embargo, ambos métodos requieren que las partes que se están comunicando dispongan de una forma de establecer un canal seguro. En otros términos, ya debe haber ocurrido alguna forma de establecimiento y distribución de claves. El mismo argumento es aplicable cuando se establece una clave secreta compartida por medio de una tercera parte confiable, tal como un KDC.

Un esquema elegante y ampliamente utilizado para establecer una clave compartida a través de un canal inseguro es el **intercambio de claves de Diffie-Hellman** (Diffie y Hellman, 1976). El protocolo funciona como sigue. Supongamos que Alicia y Bob desean establecer una clave secreta compartida. El primer requerimiento es que se pongan de acuerdo en dos números grandes n y g sujetos a varias propiedades matemáticas (las cuales no se analizan aquí). Tanto n como g pueden hacerse públicos; no es necesario ocultarlos de los extraños. Alicia elige un número aleatorio grande, por ejemplo x, el cual mantiene secreto. Asimismo, Bob selecciona su propio número grande secreto, digamos, y. En este momento existe suficiente información para construir una clave secreta, como se muestra en la figura 9-33.

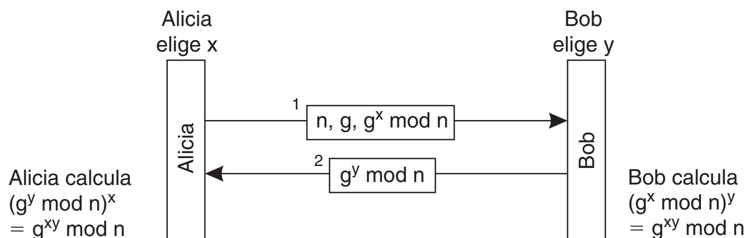


Figura 9-33. El principio de intercambio de claves de Diffie-Hellman.

Alicia comienza enviando g^x *mod* n a Bob, junto con n y g. Es importante señalar que esta información puede ser enviada como texto común, ya que es virtualmente imposible calcular x

dado g^x *mod n*. Cuando Bob recibe el mensaje, posteriormente calcula $(g^x \text{ mod } n)^y$ que matemáticamente es igual a g^{xy} *mod n*. Además, envía g^y *mod n* a Alicia, que luego puede calcular $(g^y \text{ mod } n)^x = g^{xy} \text{ mod } n$. Por consiguiente, tanto Alicia como Bob, y sólo ellos, tendrán ahora en su poder la clave secreta compartida g^{xy} *mod n*. Observemos que ninguno de ellos tiene que revelar su número privado (x y y, respectivamente) al otro.

Diffie-Hellman puede ser considerado como un criptosistema de clave pública. En el caso de Alicia, x es su clave privada, en tanto que g^x *mod n* es su clave pública. Como se verá a continuación, la distribución segura de la clave pública es esencial para que Diffie-Hellman funcione en la práctica.

Distribución de claves

Una de las partes más difíciles de realizar en la administración de claves es la distribución de las claves iniciales. En un criptosistema simétrico, la clave secreta compartida inicial debe ser comunicada a través de un canal seguro que proporcione autenticación y confidencialidad, como se muestra en la figura 9-34(a). Si Alicia y Bob no disponen de claves para establecer el canal seguro, es necesario distribuir fuera de banda la clave no disponible. En otros términos, Alicia y Bob tendrán que ponerse en contacto entre sí con algún otro medio de comunicación que no sea la red. Por ejemplo, uno de ellos puede telefonear al otro, o enviar la clave en un disco flexible a través del correo convencional.

En el caso de un criptosistema de clave pública, la clave pública tiene que ser distribuida en tal forma que los destinatarios estén seguros de que sí está pareada con una clave secreta. En otros términos, como se muestra en la figura 9-34(b), aunque la clave pública puede ser enviada como texto común, es necesario que el canal a través del cual es enviada pueda proporcionar autenticación. La clave privada, desde luego, tiene que ser enviada a través de un canal seguro que proporcione autenticación y confidencialidad.

Cuando se trata de distribución de claves, la distribución autenticada de claves públicas es tal vez lo más interesante. En la práctica, la distribución de una clave pública ocurre por medio de **certificados de clave pública**. Tales certificados consisten en una clave pública junto con una cadena que identifica la entidad con la cual dicha clave está asociada. La entidad podría ser un usuario, aunque también un servidor o algún dispositivo especial. Una **autoridad certificadora** firma la clave pública y el identificador, y la firma también se incluye en el certificado. (La identidad de la autoridad certificadora, naturalmente, no forma parte de la autoridad certificadora.) La firma se realiza por medio de una clave privada K_{CA}^- que pertenece a la autoridad certificadora. Se supone que la clave pública correspondiente K_{CA}^+ es conocida. Por ejemplo, las claves públicas de varias autoridades certificadoras vienen incluidas en la mayoría de los navegadores de la web y se envían junto con los binarios.

El uso de un certificado de clave pública funciona como sigue. Supongamos que un cliente desea asegurarse de que la clave pública encontrada en el certificado pertenece realmente a la entidad identificadora. El cliente utiliza entonces la clave pública de la autoridad certificadora asociada para verificar la firma del certificado. Si la firma que aparece en el certificado concuerda

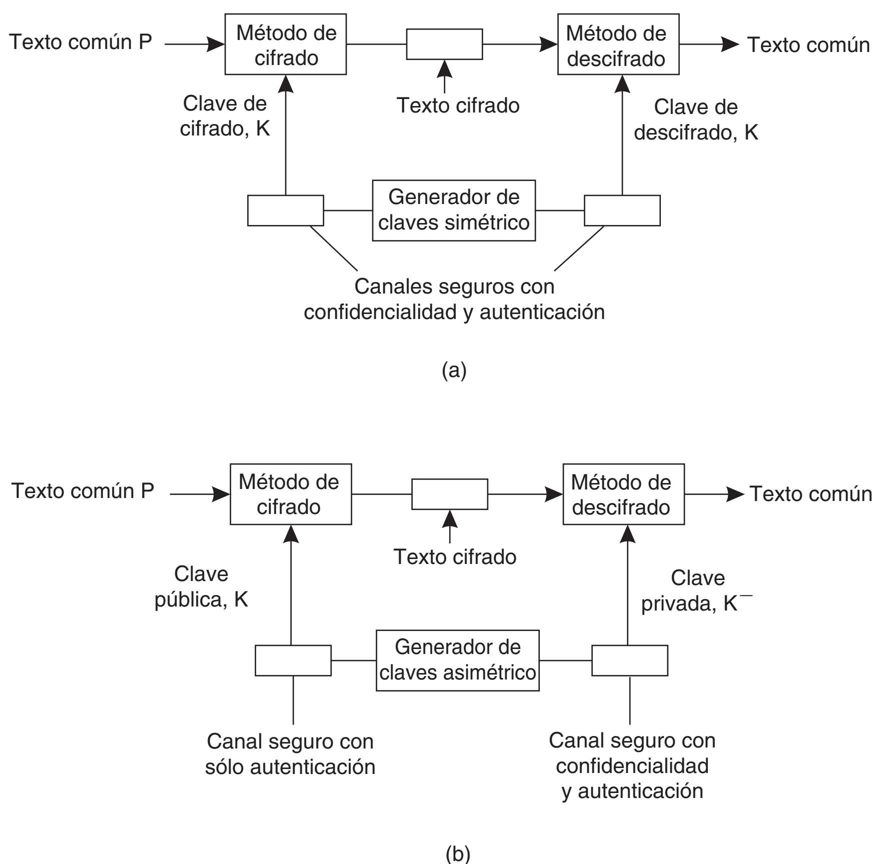


Figura 9-34. (a) Distribución de claves secretas. (b) Distribución de claves públicas [vea también Menezes y cols., (1996)].

con el par (*clave pública*, *identificador*), el cliente acepta que la clave pública pertenece a la entidad identificada.

Es importante señalar que cuando se acepta que el certificado está en orden, el cliente en realidad confía en que no ha sido falsificado. En particular, el cliente debe asumir que la clave pública K_{CA}^{+} sí pertenece a la autoridad certificadora asociada. En caso de duda, deberá ser posible verificar la validez de K_{CA}^{+} mediante otro certificado que provenga de una autoridad certificadora diferente, quizá más confiable.

Tales **modelos confiables** jerárquicos, donde todos deben confiar en la autoridad certificadora de más alto nivel, no son comunes. Por ejemplo, el **PEM** (*Privacy Enhanced Mail*; **correo de privacidad mejorada**) utiliza un modelo confiable de tres niveles en el que las autoridades certificadoras de más bajo nivel pueden ser autenticadas por **PCA** *(Policy Certification Authorities;* **autoridades certificadoras de políticas**), las que a su vez pueden ser autenticadas por la **IPRA** (*Internet Policy*

Registration Authority; **autoridad de registro de políticas en internet**). Si un usuario no confía en la IPRA, o piensa que no es seguro hablar con ella, no es de esperarse que algún día confíe en que los mensajes de correo electrónico confiables sean enviados de una forma segura cuando se utiliza el PEM. Más información sobre este modelo puede consultarse en Kent (1993). En Menezes y colaboradores (1996) se analizan otros modelos confiables.

Duración de los certificados

Un tema importante relacionado con los certificados es su longevidad. En primer término, consideremos la situación en que una autoridad certificadora entrega certificados de por vida. En esencia, lo que el certificado manifiesta es que la clave pública siempre será válida para la identidad identificada por el certificado. Claramente, esta manifestación no es deseable. Si alguna vez se compromete la clave privada de la entidad identificada, un cliente poco sospechoso jamás deberá ser habilitado para utilizar la clave pública (mucho menos los clientes maliciosos). En ese caso, se requiere de un mecanismo últil para **revocar** el certificado haciendo que se sepa que ya no es válido.

Existen varias formas de revocar un certificado. Un método común es con una **CRL** (*Certificate Revocation List*; **lista de revocación de certificado**) publicada con regularidad por la autoridad certificadora. Siempre que un cliente verifique un certificado, tendrá que consultar la CRL para ver si ha sido o no revocado. Esto significa que el cliente deberá ponerse en contacto con la autoridad certificadora, por lo menos, cada vez que se publique una CRL nueva. Observemos que la CRL se publica a diario, también se requiere un día para revocar un certificado. Mientras tanto, un certificado comprometido puede ser utilizado en forma dolosa hasta ser publicado en la siguiente CRL. En consecuencia, el tiempo entre la publicación de las CRL no debe ser demasiado largo. Además, la obtención de una CRL presupone algunos gastos indirectos.

Un método alternativo es restringir la duración de cada certificado. La validez de un certificado expira automáticamente después de cierto tiempo. Si por cualquier razón el certificado debe revocarse antes de que expire, la autoridad certificadora aún lo puede publicar en una CRL. Sin embargo, este método seguirá obligando a que los clientes consulten la CRL más reciente siempre que verifiquen un certificado. En otros términos, tendrán que ponerse en contacto con la autoridad certificadora o con una base de datos confiable que contenga la CRL más reciente.

Un caso final extremo es reducir la duración de un certificado a casi cero. En realidad, esto significa que no se utilizarán certificados; en su lugar, un cliente siempre tendrá que ponerse en contacto con la autoridad certificadora para verificar la validez de una clave pública. En consecuencia, la autoridad certificadora debe estar continuamente en línea.

En la práctica, los certificados se entregan con duraciones restringidas. En el caso de aplicaciones de internet, el tiempo de expiración a menudo es de más de un año (Stein, 1998). Tal método requiere que las CRL se publiquen con regularidad, pero que también sean inspeccionadas cuando se verifiquen certificados. La práctica indica que las aplicaciones de cliente rara vez consultan las CRL, y simplemente asumen que un certificado es válido hasta que expira. A este respecto, cuando se trata de seguridad en internet en la práctica, sigue habiendo mucho por mejorar, desafortunadamente.

9.4.2 Administración de un grupo seguro

Muchos sistemas de seguridad utilizan servicios especiales como los KDC (*Key Distribution Centers*; centros de distribución de claves) o las CA (*Certification Authorities*; autoridades certificadoras). Estos servicios ponen de manifiesto una dificultad presente en los sistemas distribuidos. Por principio de cuentas, deben ser confiables. Para acrecentar la confianza en los servicios de seguridad, es necesario proporcionar un alto grado de protección contra toda clase de amenazas. Por ejemplo, tan pronto como una CA ha sido comprometida, se vuelve imposible constatar la validez de una clave pública, ello hace que todo el sistema de seguridad sea completamente inútil.

Por otra parte, también es necesario que muchos servicios de seguridad ofrezcan una disponibilidad elevada. Por ejemplo, en el caso de un KDC, cada vez que dos procesos deseen establecer un canal seguro, por lo menos uno de ellos tendrá que ponerse en contacto con el KDC en busca de una clave compartida. Si el KDC no está disponible, no se puede establecer una comunicación segura a menos que esté disponible una técnica alternativa de establecimiento de claves, tal como el intercambio de clave de Diffie-Hellman.

La solución para lograr una disponibilidad elevada es la replicación. Por otra parte, ésta propicia que un servidor sea más vulnerable a ataques de seguridad. Ya estudiamos cómo se puede lograr la comunicación segura de un grupo cuando sus miembros comparten un secreto. En realidad, ningún miembro del grupo es capaz de comprometer certificados, ello hace que el propio grupo sea altamente seguro. Lo que resta considerar es cómo se administra realmente un grupo de servidores replicados. Reiter y colaboradores (1994) proponen la siguiente solución.

El problema a resolver es garantizar que cuando un proceso solicite unirse a un grupo G, la integridad de éste no se comprometa. Se supone que un grupo G utiliza un clave secreta CK_G compartida por todos sus miembros para cifrar sus mensajes. Además, también utiliza un par de claves pública-privada K_G^+, K_G^- para comunicarse con miembros ajenos al grupo.

Siempre que un proceso P desea unirse a un grupo G, envía una petición de unión JR que identifica a G y P, el tiempo local de P como T, una *nota de respuesta* RP, y una clave secreta generada $K_{P,G}$. RP y $K_{P,G}$ se cifran con la clave pública del grupo K_G^+, como se muestra en el mensaje 1 de la figura 9-35. El uso de RP y $K_{P,G}$ se explica con más detalle a continuación. La respuesta JR es firmada por P y se envía junto con un certificado que contiene la clave pública de P. Se ha utilizado ampliamente la notación $[M]_A$ para denotar que el mensaje M fue firmado por el sujeto A.

Cuando un miembro del grupo Q recibe esa respuesta, primero auténtica P, tras de lo cual se comunica con los demás miembros del grupo para ver si P puede ser admitido como miembro. La autenticación de P ocurre de la manera usual por medio del certificado. La marca de tiempo T se utiliza para asegurarse de que el certificado aún era válido en el momento en que fue enviado. (Observemos que es necesario asegurarse de que el tiempo no ha sido manipulado.) El miembro del grupo Q verifica la firma de la autoridad certificadora y posteriormente extrae del certificado la clave pública de P para verificar la validez de JR. En ese momento, se sigue un protocolo específico del grupo para ver si todos sus miembros están de acuerdo en admitir a P.

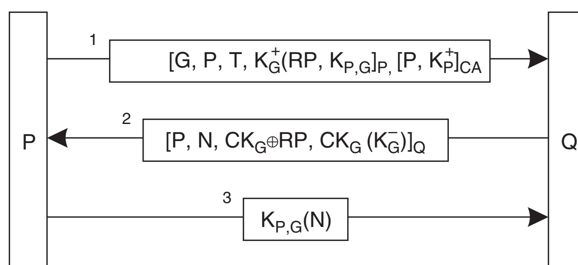


Figura 9-35. Admisión segura de un nuevo miembro del grupo.

Si se permite que *P* se una al grupo, *Q* regresa un mensaje de admisión *GA*, mostrado como el mensaje 2 en la figura 9-35, que identifica a *P* y contiene un nonce *N*. La nota *RP* de respuesta se utiliza para cifrar la clave de comunicación del grupo CK_G. Además, *P* también va a necesitar la clave privada del grupo K_G^-, la cual se cifra con CK_G. Luego *Q* firma el mensaje *GA* con la clave $K_{P,G}$.

El proceso *P* ahora puede autenticar a *Q*, porque sólo un verdadero miembro del grupo pudo haber descubierto la clave secreta $K_{P,G}$. En este protocolo no se utiliza el nonce *N* por seguridad; en su lugar, cuando *P* regresa *N* cifrado con $K_{P,G}$ (mensaje 3), *Q* en ese momento sabe que *P* recibió todas las claves necesarias, y que por consiguiente ahora si se unió al grupo.

Observemos que en lugar de utilizar la nota de respuesta *RP*, *P* y *Q* también podían haber cifrado CK_G con la clave pública de *P*. Sin embargo, como *RP* se utiliza sólo una vez; es decir, para cifrar la clave de comunicación del grupo en el mensaje *GA*, es más seguro utilizar *RP*. Si la clave privada de *P* fuera revelada alguna vez, también llegaría a ser posible revelar CK_G, lo cual comprometería la seguridad de toda la comunicación del grupo.

9.4.3 Administración de la autorización

La administración de la seguridad en los sistemas distribuidos también tiene que ver con la administración de los derechos de acceso. Hasta ahora, apenas se ha tocado el tema de cómo se otorgan inicialmente los derechos de acceso a usuarios o grupos de usuarios y cómo, posteriormente, se mantienen inviolables. Es tiempo de corregir esta omisión.

En sistemas no distribuidos, administrar los derechos de acceso es relativamente fácil. Cuando se agrega un nuevo usuario al sistema, se le otorgan derechos iniciales como, por ejemplo, crear archivos y subdirectorios en un directorio específico, crear procesos, utilizar tiempo de CPU, y así sucesivamente. En otros términos, se abre una cuenta para el usuario en una máquina específica donde todos los derechos han sido especificados con anticipación por los administradores del sistema.

En un sistema distribuido, las cosas se complican por el hecho de que los recursos están ubicados en varias máquinas. Si se tuviera que seguir el método para sistemas no distribuidos, sería necesario crear una cuenta para cada usuario en cada máquina. En esencia, éste es el método que se sigue en sistemas operativos de red. Las cosas se simplifican un poco al crear una cuenta única en

un servidor central. Este servidor es consultado cada vez que un usuario accede a ciertos recursos o máquinas.

Capacidades y certificados de atributo

Un método mucho mejor, y que se aplica ampliamente en sistemas distribuidos, es el uso de capacidades. Como brevemente se explicó con anterioridad, una **capacidad** es una estructura de datos infalsificable, para un recurso determinado, que especifica con exactitud los derechos de acceso que el portador de la capacidad tiene con respecto a dicho recurso. Existen diferentes implementaciones de capacidades. Aquí se analiza brevemente la implementación utilizada en el sistema operativo Amoeba (Tanenbaum y cols., 1986).

Amoeba fue uno de los primeros sistemas distribuidos basados en objetos. Su modelo de objetos distribuidos es el de objetos remotos. En otros términos, un objeto reside en un servidor mientras que a los clientes se les ofrece acceso transparente a dicho objeto mediante un proxy. Para invocar una operación sobre un objeto, un cliente transfiere una capacidad a su sistema operativo local, el que luego localiza el servidor donde reside el objeto y, posteriormente, hace una llamada de proceso remoto (RPC) a dicho servidor.

Una capacidad es un identificador de 128 bits organizado internamente como se muestra en la figura 9-36. Los primeros 48 bits son inicializados por el servidor del objeto cuando éste es creado y, efectivamente, forman un identificador del servidor del objeto independiente de la máquina, conocido como **puerto del servidor**. Amoeba utiliza transmisión para ubicar la máquina donde el servidor está actualmente localizado.

48 bits	24 bits	8 bits	48 bits
Puerto de servidor	Objeto	Derechos	Verificación

Figura 9-36. Una capacidad en Amoeba.

Los siguientes 24 bits se utilizan para identificar el objeto en el servidor dado. Observemos que en Amoeba el puerto del servidor junto con el identificador del objeto forman un identificador único de 72 bits a nivel de todo el sistema por cada objeto. Los siguientes 8 bits se utilizan para especificar los derechos de acceso del portador de la capacidad. Finalmente, se utiliza el campo de *verificación* de 48 bits para hacer que la capacidad sea infalsificable, tal como se explica en las páginas siguientes.

Cuando se crea un objeto, su servidor elige al azar un campo de *verificación* y lo guarda tanto en la capacidad como internamente en sus propias tablas. Inicialmente todos los bits de derechos se encuentran en una capacidad y esta **capacidad del propietario** es la que se regresa al cliente. Cuando la capacidad es enviada de regreso al servidor en una petición para realizar una operación, se comprueba el campo de *verificación*.

Para crear una capacidad restringida, un cliente puede regresar una capacidad al servidor junto con una máscara de bits para los nuevos derechos. El servidor toma el campo de *verificación* original de sus tablas, le aplica la operación XOR con los nuevos derechos (los cuales deben ser un subcon-

junto de los derechos presentes en la capacidad), y luego ejecuta el resultado mediante una función unidireccional.

El servidor crea entonces una nueva capacidad con el mismo valor que aparece en el campo *objeto*, pero con los nuevos derechos en el campo *derechos* y la salida de la función unidireccional en el campo *verificación*. La nueva capacidad es regresada entonces al que llama. El cliente puede enviar esta nueva capacidad a otro proceso, si lo desea.

El método de generar capacidades restringidas se ilustra en la figura 9-37. En este ejemplo, el propietario ha renunciado a todos los derechos excepto a uno. Por ejemplo, la capacidad restringida podría permitir leer el objeto, pero nada más. El significado del campo *derechos* es diferente para cada tipo de objeto puesto que las propias operaciones legales pueden variar de un tipo de objeto a otro.

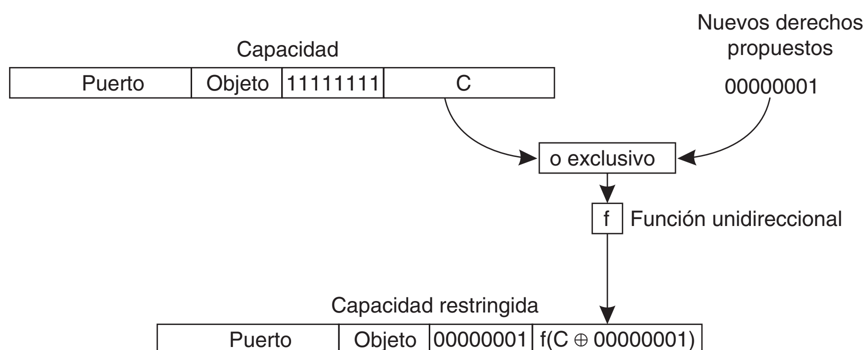


Figura 9-37. Generación de una capacidad restringida a partir de una capacidad de propietario.

Cuando la capacidad restringida regresa al servidor, éste ve desde el campo *derechos* que no es una *capacidad de propietario* porque al menos un bit no aparece. El servidor toma luego el número aleatorio original de sus tablas, le aplica la operación XOR con el campo *derechos* de la capacidad, y ejecuta el resultado con la función unidireccional. Si el resultado coincide con el campo *verificación*, la capacidad es aceptada como válida.

Deberá resultar evidente a partir de este algoritmo que el usuario que pretende agregar derechos que no tiene simplemente invalidará la capacidad. Invertir el campo *verificación* en una capacidad restringida para obtener el argumento (C XOR 00000001 en la figura 9-37) es imposible porque la función f es unidireccional. Es mediante esta técnica criptográfica que las capacidades se protegen contra manipulación. Observemos que esencialmente f calcula un resumen de mensaje, tal como se vio con anterioridad. Cualquier cambio en el mensaje original (como la inversión de un bit), será detectado de inmediato.

Una generalización de capacidades que en ocasiones se utiliza en los sistemas distribuidos modernos es el **certificado de atributo**. A diferencia de los certificados ya vistos que se utilizan para validar una clave pública, los certificados de atributo se utilizan para poner en lista ciertos pares (*atributo*, *valor*) aplicables a una entidad identificada. En particular, se pueden utilizar certi-

ficados de atributo para enumerar los derechos de acceso que el portador de un certificado tiene con respecto al recurso identificado.

Al igual que otros certificados, los de atributos son entregados por autoridades certificadoras especiales, llamadas en general **autoridades certificadoras de atributo**. Comparada con las capacidades de Amoeba, tal autoridad corresponde al servidor de un objeto. En general, sin embargo, la autoridad certificadora de atributo y el servidor que administra la entidad para la cual se creó el certificado no tiene que ser la misma. Los derechos de acceso que aparecen en un certificado son firmados por la autoridad certificadora de atributo.

Delegación

Ahora consideremos el siguiente problema. Un usuario desea imprimir un gran archivo para el cual tiene derechos de sólo lectura. Para no molestar demasiado a otros, el usuario envía una solicitud al servidor de impresión y le pide que comience a imprimir el archivo no antes de las dos de la mañana. En lugar de enviar todo el archivo a la impresora, el usuario transfiere el nombre del archivo a la impresora de modo que pueda ponerlo en la cola de impresión, si es necesario, cuando realmente se requiera.

Aunque este esquema parece estar perfectamente en orden, existe un problema importante: por lo general, la impresora no tendrá los permisos de acceso apropiados al archivo mencionado. Es decir, si no se toman ningunas medidas especiales, en cuanto el servidor de impresión pretenda leer el archivo para imprimirlo, el sistema le negará el acceso al archivo. Este problema podría haber sido resuelto si el usuario hubiera **delegado** temporalmente al servidor de impresión sus derechos de acceso al archivo.

La delegación de derechos de acceso es una técnica importante de protección de sistemas de computadora y de sistemas distribuidos, en particular. La idea básica es simple: traspasando ciertos derechos de acceso de un proceso a otro, se facilita la distribución del trabajo entre varios procesos sin afectar adversamente la protección de los recursos. En el caso de sistemas distribuidos, los procesos pueden ser ejecutados en máquinas diferentes e incluso dentro de dominios administrativos diferentes, como en el caso de Globus. La delegación evita muchos gastos indirectos ya que a menudo la protección puede ser manejada localmente.

Existen varias formas de implementar la delegación. Un método general, como se describe en Neuman (1993), es utilizar un proxy. En el contexto de seguridad en sistemas de computadora, un **proxy** es un token que permite a su propietario operar con los mismos o con restringidos derechos y privilegios del sujeto que otorgó el token. (Observemos que esta noción de un proxy es diferente de un proxy como sinónimo de módulo o adaptador del lado del cliente. Aunque tratamos de evitar sobrecargar los términos, aquí se hace una excepción ya que en la definición anterior el término "proxy" es muy ampliamente utilizado como para ignorarlo.) Un proceso puede crear un proxy con, cuando mucho, los mismos derechos y privilegios que él tiene. Si un proceso crea un proxy nuevo basado en el que actualmente tiene, el proxy derivado tendrá por lo menos las mismas restricciones que el original, y posiblemente más.

Antes de considerar un esquema general de delegación, consideremos los dos métodos siguientes. En primer lugar, la delegación es relativamente simple si Alicia conoce a todos. Si desea delegar derechos a Bob, simplemente necesita construir un certificado que diga "Alicia dice que Bob tiene

derechos *R*", tales como $[A,B,R]_A$. Si Bob desea traspasar algunos de sus derechos a Charlie, le pedirá a éste que se ponga en contacto con Alicia y le pida un certificado apropiado.

En un sencillo segundo caso, Alicia puede construir simplemente un certificado que diga "El portador de este certificado tiene derechos *R*". Sin embargo, en esta ocasión se tiene que proteger al certificado contra copiado ilegal, como se hace en el traspaso seguro de capacidades entre procesos. El esquema de Neuman maneja este caso, y también evita el problema de que Alicia tenga que conocer a todos aquellos a quienes los derechos tengan que ser delegados.

En el esquema de Neuman un proxy consta de dos partes, como se ilustra en la figura 9-38. Sea *A* el proceso que creó el proxy. La primera parte de éste es un conjunto $C = \{R, S^+_{proxy}\}$, compuesto a partir de un conjunto *R* de derechos de acceso que han sido delegados por *A*, junto con una parte públicamente conocida de un secreto utilizado para autenticar al portador del certificado. Se explicará el uso de S^+_{proxy} a continuación. El certificado porta la firma $sig(A,C)$ de *A*, para protegerlo contra modificaciones. La segunda parte contiene la otra parte del secreto, denotada como S^-_{proxy}. Es esencial que S^-_{proxy} esté protegida contra revelación cuando se deleguen derechos a otro proceso.

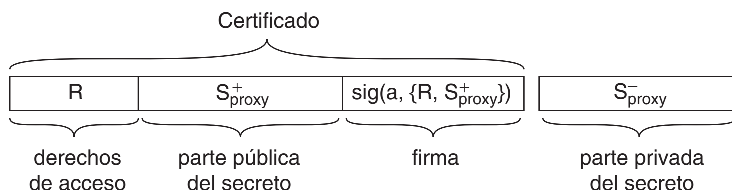


Figura 9-38. Estructura general de un proxy tal como se utiliza para delegación.

Otra forma de mirar el proxy es como sigue. Si Alicia desea delegar algunos de sus derechos a Bob, hace una lista de derechos (*R*) que Bob puede ejercer. Firmando la lista, impide que Bob la manipule. Sin embargo, tener sólo una lista de derechos firmada a menudo no es suficiente. Si Bob desea ejercer sus derechos, es posible que deba demostrar que realmente obtuvo la lista de Alicia y que, por ejemplo, no se la robó a alguien más. Por consiguiente, Alicia se saca de la manga una maliciosa pregunta (S^+_{proxy}) de la cual sólo ella conoce la respuesta a (S^-_{proxy}). Cualquiera puede verificar con facilidad la corrección de la respuesta cuando se hace la pregunta. La pregunta se anexa a la lista antes de que Alicia agregue su firma.

Cuando delega algunos de sus derechos, Alicia entrega a Bob la lista de derechos firmada, junto con la maliciosa pregunta. También le proporciona la respuesta asegurándole que nadie puede interceptarla. Bob tiene entonces una lista de derechos, firmada por Alicia, la cual puede entregar a, por ejemplo, Charlie, cuando se requiera. Charlie le hará la maliciosa pregunta que aparece al final de la lista. Si Bob conoce la respuesta, Charlie sabrá con seguridad que Alicia sí había delegado a Bob los derechos listados.

Una propiedad importante de este esquema es que Alicia no tiene que ser consultada. En realidad, Bob puede decidir si traspasa (algunos) de los derechos que aparecen en la lista a Dave. Al hacerlo, también le dirá a Dave la respuesta a la pregunta, de modo que Dave pueda demostrar que

la lista le fue dada a él por alguien que tenía derecho a ella. Alicia no necesita saber nada sobre Dave en absoluto.

Un protocolo para delegar y ejercer derechos se muestra en la figura 9-39. Supongamos que Alicia y Bob comparten una clave secreta $K_{A,B}$ que puede ser utilizada para cifrar mensajes que se envían entre sí. Entonces, Alicia primero envía a Bob el certificado $C = \{R, S^{+}_{proxy}\}$, firmado con $sig(A,C)$ y denotado de nuevo como $[R, S^{+}_{proxy}]_A$). No es necesario cifrar este mensaje: puede ser enviado como texto común. Sólo tiene que ser cifrada la parte privada del secreto, mostrada como $K_{A,B}(S^{-}_{proxy})$ en el mensaje 1.

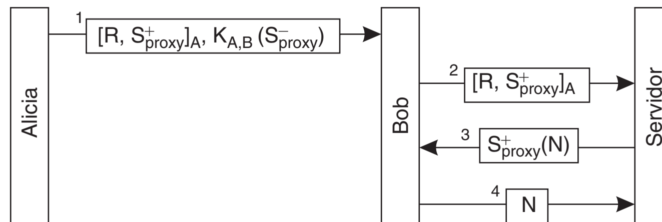


Figura 9-39. Utilización de un proxy para delegar y comprobar los derechos de acceso.

Ahora supongamos que Bob desea realizar una operación en un objeto residente en un servidor específico. También, supongamos que Alicia está autorizada para hacer que se realice la operación y que ha delegado sus derechos a Bob. Por consiguiente, Bob entrega sus credenciales al servidor en la forma del certificado firmado $[R, S^{+}_{proxy}]_A$.

En ese momento, el servidor verificará que C no haya sido manipulado: cualquier modificación a la lista de derechos, o a la pregunta maliciosa será advertida, porque ambas fueron firmadas por Alicia. No obstante, el servidor aún no sabe si Bob es el propietario legítimo del certificado. Para verificarlo, el servidor debe utilizar el secreto que llegó con C.

Existen varias formas de implementar (S^{+}_{proxy}) y S^{-}_{proxy}. Por ejemplo, supongamos que (S^{+}_{proxy}) es una clave pública y que S^{-}_{proxy} es la clave privada correspondiente. Z puede entonces retar a Bob enviándole un nonce N, cifrado con S^{+}_{proxy}. Cifrando $S^{+}_{proxy}(N)$ y regresando N, Bob demuestra que conoce el secreto y que, por tanto, es el portador legítimo del certificado. Existen otras formas de implementar la delegación segura, aunque la idea básica siempre es la misma: demostrar que se conoce un secreto.

9.5 RESUMEN

La seguridad desempeña un rol extremadamente importante en los sistemas distribuidos. Un sistema distribuido deberá proporcionar mecanismos que permitan aplicar varias políticas de seguridad diferentes. El desarrollo y la aplicación apropiada de dichos mecanismos generalmente hacen que la seguridad sea un difícil ejercicio de ingeniería.

Se distinguen tres temas importantes. El primero es que un sistema distribuido deberá ofrecer los medios necesarios para establecer canales seguros entre procesos. Un canal seguro, en principio, proporciona los medios para autenticar manualmente las partes que se están comunicando y para proteger los mensajes contra modificaciones durante su transmisión. Un canal seguro, en general, proporciona confidencialidad de tal forma que nadie más que la partes en comunicación puedan leer los mensajes transmitidos por el canal.

Un tema de diseño importante es si utilizar sólo un criptosistema simétrico (basado en claves secretas compartidas) o combinarlo con un sistema de clave pública. La práctica actual utiliza criptografía de clave pública para distribuir claves secretas compartidas de corta duración. Las últimas se conocen como claves de sesión.

El segundo tema en sistemas distribuidos seguros es el control de acceso, o autenticación. La autorización se ocupa de proteger los recursos de tal forma que sólo aquellos procesos que tengan los derechos de acceso apropiados puedan acceder a ellos y utilizarlos. El control de acceso siempre ocurre después de que un proceso ha sido autenticado. Relacionado con el control de acceso está evitar la negación de servicio, lo cual se torna en un problema difícil en sistemas que son accesibles a través de internet.

Existen dos formas de implementar el control de acceso. Primero, cada recurso puede llevar una lista de control de acceso que incluya con exactitud los derechos de acceso de cada usuario o proceso. Alternativamente, un proceso puede portar un certificado que establezca con precisión cuáles son sus derechos para un conjunto particular de recursos. El beneficio principal de utilizar certificados es que un proceso puede transferir con facilidad su boleto a otro proceso, es decir, delegar sus derechos de acceso. Los certificados, sin embargo, tienen la desventaja de que con frecuencia son difíciles de revocar.

Se requiere atención especial con el control de acceso en el caso de un código móvil. Además, proteger un código móvil contra un servidor malicioso, en general, es más importante que protegerlo contra un código móvil malicioso. Se han hecho varias proposiciones, de las cuales la caja de arena es actualmente la más aplicada. Sin embargo, las cajas de arena son algo restrictivas y por ello se han ideado métodos más flexibles basados en dominios de protección verdadera.

El tercer tema en los sistemas distribuidos seguros tiene que ver con administración. Existen esencialmente dos subtemas importantes: la administración de claves y la administración de la autorización. La administración de claves incluye la distribución de claves criptográficas, para lo cual los certificados emitidos por terceras partes confiables desempeñan un rol importante. Con respecto a la administración de la autorización, también son importantes los certificados de atributo y la delegación.

PROBLEMAS

1. ¿Qué mecanismos podría proporcionar un sistema distribuido como servicios de seguridad a los desarrolladores que creen sólo en el argumento fin a fin en el diseño de sistemas, tal como se vio en el capítulo 6?

2. En el método RISSC, ¿puede o no concentrarse toda la seguridad en servidores seguros?

3. Supongamos que a usted le pidieron desarrollar una aplicación distribuida que permita a los profesores elaborar exámenes. Proporcione por lo menos tres enunciados que formarían parte de la política de seguridad en dicha aplicación.

4. ¿Sería seguro unir los mensajes 3 y 4 en el protocolo de autenticación mostrado en la figura 9-12 en la forma $K_{A,B}(R_B,R_A)$?

5. ¿Por qué en la figura 9-15 no es necesario que el KDC sepa con seguridad que está hablando con Alicia cuando recibe una solicitud de una clave secreta que Alicia puede compartir con Bob?

6. ¿Por qué es erróneo implementar un nonce como marca de tiempo?

7. En el mensaje 2 del protocolo de autenticación de Needham-Schroeder, el boleto se cifra con la clave secreta compartida por Alicia y el KDC, ¿es necesario este cifrado?

8. ¿Puede adaptarse con seguridad el protocolo de autenticación mostrado en la figura 9-19 de modo que el mensaje 3 se componga sólo de R_B?

9. Diseñe un protocolo de autenticación simple por medio de firmas en un criptosistema de clave pública.

10. Suponga que Alicia desea enviar un mensaje m a Bob. En lugar de cifrar m con la clave pública de Bob K_B^+, Alicia genera una clave de sesión $K_{A,B}$ y luego envía $[K_{A,B}(m),K_B^+(K_{A,B})]$. ¿Por qué generalmente es mejor este esquema? (*Sugerencia:* considere temas de desempeño.)

11. ¿Cuál es el rol de la marca de tiempo en el mensaje 6 de la figura 9-23 y por qué debe ser cifrada?

12. Complete la figura 9-23 agregando la comunicación de autenticación entre Alicia y Bob.

13. ¿Cómo se pueden expresar los cambios de rol en una matriz de control de acceso?

14. ¿Cómo se implementan las ACL en un sistema de archivos UNIX?

15. ¿Cómo puede una organización hacer que se utilice una compuerta proxy web para impedir que sus usuarios accedan directamente a servidores externos a la web?

16. Remitiéndose a la figura 9-31, ¿a qué grado el uso de referencias a objetos Java como capacidades realmente depende del lenguaje Java?

17. Mencione tres problemas que se presentan cuando se requiere que los desarrolladores de interfaces para recursos locales inserten llamadas para habilitar o deshabilitar privilegios como protección contra el acceso no autorizado de programas móviles, tal como se explicó en el texto.

18. Mencione algunas ventajas y desventajas de utilizar servidores centralizados para administrar claves.

19. El protocolo de intercambio de claves de Diffie-Hellman también puede ser utilizado para establecer una clave secreta compartida entre partes. Explique cómo se logra esto.

20. No hay autenticación en el protocolo de intercambio de claves de Diffie-Hellman. Explotando esta propiedad, una tercera parte maliciosa, Chuck, puede irrumpir con facilidad en el intercambio de claves que está ocurriendo entre Alicia y Bob, y posteriormente arruinar la seguridad. Explique cómo funcionaría esto.

21. Proporcione una forma simple de cómo pueden ser revocadas las capacidades en Amoeba.

22. ¿Tiene sentido restringir la duración de una clave de sesión? De ser así, proporcione un ejemplo de cómo podría establecerse.

23. (**Asignación para el laboratorio.**) Instale y configure un ambiente Kerberos v5 para un sistema distribuido integrado por tres máquinas diferentes. Una de las máquinas deberá ejecutar el KDC. Asegúrese de que puede configurar una conexión telnet (Kerberos) entre dos máquinas cualesquiera, pero utilizando sólo una contraseña registrada en el KDC. Muchos de los detalles sobre la ejecución de Kerberos se explican en Garman (2003).

10

SISTEMAS BASADOS EN OBJETOS DISTRIBUIDOS

Con este capítulo se pasa del análisis de los principios a un examen de los diversos paradigmas utilizados para organizar sistemas distribuidos. El primer paradigma se compone de objetos distribuidos. En sistemas basados en objetos distribuidos, la noción de un objeto desempeña un rol fundamental al establecer la transparencia de la distribución. Por principio, cualquier cosa se trata como un objeto y a los clientes se les ofrecen servicios y recursos en la forma de objetos que pueden invocar.

Los objetos distribuidos constituyen un importante paradigma porque es relativamente fácil ocultar los aspectos de distribución detrás de la interfaz de un objeto. Además, como virtualmente un objeto puede ser cualquier cosa, también es un poderoso paradigma para construir sistemas. En este capítulo veremos cómo se aplican los principios de los sistemas distribuidos a varios sistemas basados en objetos bien conocidos. En particular, se cubren aspectos de CORBA, sistemas basados en Java y Globe.

10.1 ARQUITECTURA

La orientación a objetos constituye un importante paradigma en el desarrollo de software. Desde su presentación, ha disfrutado de una enorme popularidad. Esta popularidad se deriva de la habilidad natural de construir software en la forma de componentes independientes más o menos bien definidos. Algunos desarrolladores podrían concentrarse en implementar una funcionalidad específica independientemente de otros desarrolladores.

La orientación a objetos comenzó a ser utilizada para desarrollar sistemas distribuidos en los años de 1980. De nueva cuenta, la noción de un objeto independiente alojado en un servidor remoto al mismo tiempo que se lograba un alto grado de transparencia en la distribución formó una base sólida para el desarrollo de una nueva generación de sistemas distribuidos. En esta sección, primero examinaremos a fondo la arquitectura general de los sistemas distribuidos basados en objetos, tras de lo cual podremos ver cómo se utilizan los principios básicos en estos sistemas.

10.1.1 Objetos distribuidos

La característica fundamental de un objeto es que encapsula datos, llamado el **estado**, y las operaciones incluidas en tales datos, llamadas **métodos**. Los métodos se ponen a disposición mediante una **interfaz**. Es importante entender que no existe una forma "legal" en que un proceso pueda acceder o manipular el estado de un objeto que no sea a través de la invocación de los métodos disponibles para ello vía la interfaz del objeto. Un objeto puede implementar múltiples interfaces. Asimismo, dada una definición de interfaz, puede haber varios objetos que ofrezcan una forma de implementarla.

Esta separación entre interfaces y los objetos que las implementan resulta crucial en los sistemas distribuidos. Una estricta separación nos permite colocar las interfaces en una máquina en tanto que un objeto reside en otra máquina. Esta organización, mostrada en la figura 10-1, comúnmente se conoce como **objeto distribuido**.

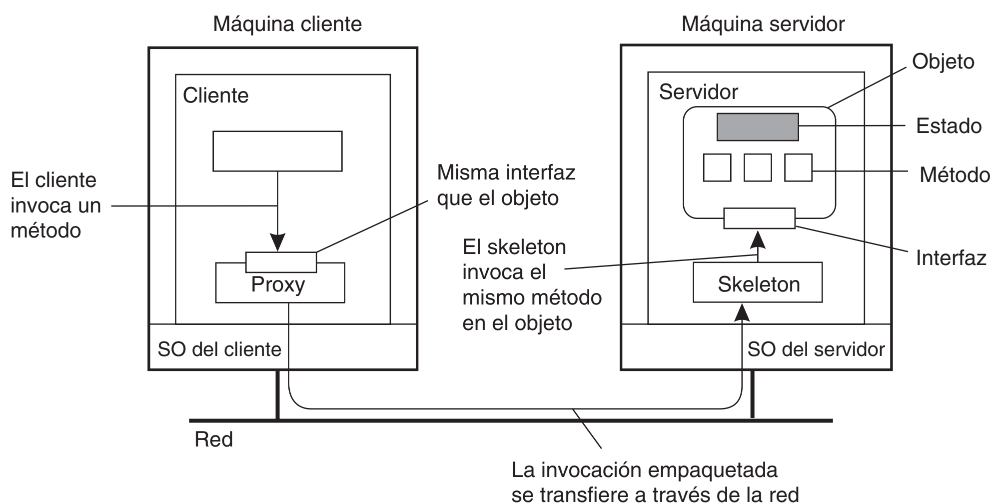


Figura 10-1. Organización común de un objeto remoto con proxy del lado del cliente.

Cuando un cliente **se vincula** a un objeto distribuido, una implementación de la interfaz de objeto, llamada **proxy**, se carga entonces en el espacio destinado a la dirección del cliente. Un proxy

es análogo a una plantilla de cliente en sistemas RPC. Lo único que hace es organizar las invocaciones a métodos en mensajes y desorganizar los mensajes de respuesta para devolver el resultado de la invocación a un método al cliente. El objeto real reside en la máquina de un servidor, donde ofrece la misma interfaz que en la máquina del cliente. Las solicitudes de invocación entrantes primero se pasan al resguardo de un servidor, el cual las desempaqueta para crear invocaciones a métodos en la interfaz del objeto ubicada en el servidor. El resguardo del servidor también es responsable de desempaquetar las respuestas y de remitir mensajes de respuesta al proxy del lado del cliente.

El resguardo del lado del cliente a menudo se conoce como plantilla de código (**skeleton**) ya que proporciona los medios básicos para permitir que el middleware del servidor acceda a los objetos definidos por el usuario. En la práctica, frecuentemente contiene un código incompleto en la forma de una clase propia de un lenguaje que tiene que ser especializado aún más por el desarrollador.

Una característica, aunque un tanto contraintuitiva de la mayoría de los objetos distribuidos es que su estado *no* es distribuido: reside en una sola máquina. Solamente las interfaces implementadas por el objeto están disponibles en otras máquinas. Esos objetos también se conocen como **objetos remotos**. En un objeto distribuido general, el estado puede estar físicamente distribuido en varias máquinas, pero su distribución también está oculta de los clientes detrás de sus interfaces.

Tiempo de compilación *versus* objetos en tiempo de ejecución

En los sistemas distribuidos, los objetos aparecen en muchas formas. La forma que resulta más evidente es la directamente relacionada con objetos a nivel de lenguaje, tales como aquellos soportados por Java, C++, u otros lenguajes orientados a objetos, los cuales se conocen como objetos en tiempo de compilación. En este caso, un objeto se define como la instancia de una clase. Una **clase** es una descripción de un tipo abstracto en función de un módulo con datos y operaciones en estos datos (Meyer, 1997).

La utilización de objetos en tiempo de compilación en sistemas distribuidos a menudo facilita la construcción de aplicaciones distribuidas. Por ejemplo, en Java, un objeto puede definirse por completo mediante su clase y las interfaces que la clase implementa. Compilando la definición de clase se obtiene un código que le permite crear una instancia real de objetos Java. Las interfaces pueden ser compiladas en resguardos del lado del cliente y del lado del servidor, lo cual permite invocar objetos Java desde una máquina remota. Un desarrollador de Java puede desconocer en gran medida la distribución de los objetos: ve sólo un código de programación Java.

La desventaja evidente de los objetos en tiempo de compilación es la dependencia de un lenguaje de programación. Por consiguiente, una forma alternativa de construir objetos distribuidos es hacerlo explícitamente durante el tiempo de ejecución. Este método se sigue en muchos sistemas distribuidos basados en objetos, ya que es independiente del lenguaje de programación en el cual están escritas las aplicaciones distribuidas. En particular, se puede construir una aplicación a partir de objetos escritos en varios lenguajes.

Cuando se trata con objetos en tiempo de ejecución, el cómo se implementan en realidad básicamente se deja abierto. Por ejemplo, un desarrollador puede decidir escribir una biblioteca C que contenga varias funciones que trabajen en un archivo de datos común. Lo esencial es cómo hacer

que esa implementación parezca ser un objeto cuyos métodos puedan ser invocados desde una máquina remota. Un método común es utilizar un **adaptador de objeto**, el cual actúa como *envolvente* alrededor de la implementación con el único propósito de darle la apariencia de un objeto. El término adaptador se deriva de un patrón de diseño descrito en Gamma y colaboradores (1994), el cual permite convertir una interfaz en algo que un cliente espera. Ejemplo de un adaptador de objeto es un objeto que se vincula dinámicamente a la biblioteca C antes mencionada y abre un archivo de datos asociado que representa el estado actual del objeto.

Los adaptadores de objeto desempeñan un rol importante en sistemas distribuidos basados en objetos. Para hacer la envoltura tan fácil como sea posible, los objetos se definen únicamente en función de las interfaces que implementan. La implementación de una interfaz puede entonces ser registrada en un adaptador, el cual posteriormente hace que la interfaz esté disponible para invocaciones (remotas). El adaptador se encargará de que las solicitudes de invocación sean atendidas y, por tanto, de proporcionar una imagen de objetos remotos a sus clientes. Más adelante en este capítulo regresamos a la organización de servidores y adaptadores de objetos.

Objetos persistentes y transitorios

Además de la distinción entre objetos a nivel de lenguaje y objetos en tiempo de ejecución, también existe una distinción entre objetos persistentes y objetos transitorios. Un **objeto persistente** es uno que continúa existiendo aun cuando ya no esté contenido en el espacio de dirección de cualquier proceso de servidor. En otros términos, un objeto persistente no depende de su servidor. En la práctica, esto significa que el servidor que actualmente está manejando el objeto persistente puede guardar su estado en un almacenamiento secundario y luego salir. Posteriormente, un servidor recién iniciado puede leer el estado del objeto en su almacenamiento y colocarlo en su propio espacio de dirección y ocuparse de solicitudes de invocación. Por contraste, un **objeto transitorio** es un objeto que existe sólo en tanto el servidor que lo está alojando exista. En cuanto el servidor deja de funcionar, el objeto deja de existir. Solía haber mucha controversia sobre la tenencia de objetos persistentes; algunas personas creen que los objetos transitorios son suficientes. Para alejar la discusión de los temas de middleware, la mayoría de los sistemas distribuidos basados en objetos simplemente soportan ambos tipos.

10.1.2 Ejemplo: Enterprise Java Beans

El lenguaje de programación Java y su modelo asociado constituyen el fundamento de numerosos sistemas distribuidos y de muchas aplicaciones. Su popularidad puede ser atribuida a que soporta la orientación a objetos, combinada con el soporte inherente para invocar métodos remotos. Como se verá más adelante en este capítulo, Java proporciona un alto grado de transparencia de acceso, por lo que es más fácil de utilizar que, por ejemplo, la combinación de C con la invocación de procedimientos remotos.

Desde su presentación, ha habido un fuerte incentivo para proporcionar los medios que faciliten el desarrollo de aplicaciones distribuidas. Estos medios van más allá del soporte de un lenguaje, ya que requieren un entorno en tiempo de ejecución que dé soporte a arquitecturas cliente-servidor

de múltiples capas o niveles tradicionales. Con este fin, se ha trabajado mucho en el desarrollo de **(Enterprise) Java Beans (EJB)**.

Un EJB es en esencia un objeto Java alojado en un servidor especial que ofrece diferentes formas para que los clientes remotos lo invoquen. Resulta crucial que este servidor permita separar la funcionalidad de la aplicación de la funcionalidad orientada al sistema. Lo segundo incluye funciones para buscar objetos, guardarlos, permitir que formen parte de una transacción, y así sucesivamente. Cómo se puede lograr esta separación lo analizamos más adelante, cuando estudiemos los servidores de objetos. El desarrollo de los EJB se describe con todo detalle en Monson-Hafael y colaboradores (2004). Las especificaciones se hallan en Sun Microsystems (2005a).

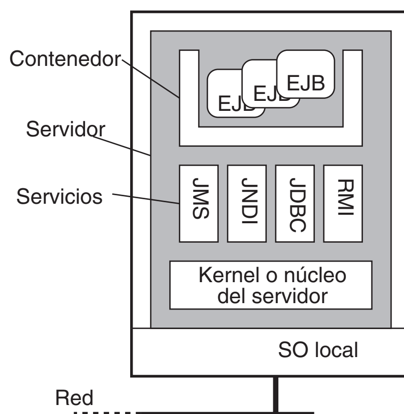


Figura 10-2. Arquitectura general de un servidor EJB (Enterprise Java Beans).

Con esta separación en mente, los EJB pueden ser ilustrados como se muestra en la figura 10-2. El tema importante es que un EJB se encuentra incrustado en el interior de un contenedor que efectivamente proporciona interfaces para los servicios subyacentes implementados por el servidor de aplicaciones. El contenedor puede vincular de una manera más o menos automática el EJB a estos servicios, lo cual significa que las referencias correctas quedan rápidamente disponibles para un programador. Los servicios típicos incluyen aquellos necesarios para la invocación de métodos remotos (RMI, del inglés *remote method invocation*), el acceso a bases de datos (JDBC), la asignación de nombres (JNDI), y la mensajería (JMS). El uso de estos servicios es más o menos automático, pero se requiere que el programador distinga entre cuatro clases de EJB:

1. Beans de sesión sin estado
2. Beans de sesión con estado
3. Beans de entidad
4. Beans dirigidos por mensajes

Como su nombre lo indica, un **bean de sesión sin estado** es un objeto transitorio que es invocado una vez, realiza su trabajo, tras de lo cual desecha cualquier información que necesitara para realizar el servicio que ofrecía a un cliente. Por ejemplo, podríamos utilizar un bean de sesión sin estado para implementar un servicio que ponga en lista los libros con la calificación más alta. En este caso, el bean estaría compuesto típicamente por una búsqueda SQL entregada a una base de datos. Los resultados se darían en un formato especial que el cliente pueda manejar, tras de lo cual su trabajo tendría que ser completado y los libros listados desecharse.

Por contraste, un **bean de sesión con estado** mantiene el estado relacionado con el cliente. El ejemplo canónico es un bean que implementa un carrito de compras electrónico como los ampliamente utilizados en el comercio electrónico. En este caso, el cliente típicamente sería capaz de poner cosas en el carro, quitarlas, y utilizar el carro para dirigirse a una caja electrónica. El bean, a su vez, típicamente accedería a bases de datos para obtener los precios actuales e información sobre varios artículos aún en existencia. Sin embargo, su vida útil seguiría siendo limitada, por lo cual se conoce como bean de sesión: cuando el cliente termina (posiblemente habiendo invocado al objeto varias veces), el bean automáticamente se destruye.

Un **bean de entidad** puede ser considerado como un objeto persistente de larga duración. Como tal, un bean de entidad se guarda generalmente en una base de datos y, de igual modo, a menudo también forma parte de transacciones distribuidas. Típicamente, los beans de entidad guardan información que puede ser necesaria la siguiente vez que un cliente específico acceda al servidor. En escenarios de comercio electrónico, se puede utilizar un bean de entidad para registrar la información de un cliente, por ejemplo, domicilio de envío, domicilio de facturación, información de una tarjeta de crédito, y así sucesivamente. En estos casos, cuando un cliente inicia una sesión, su bean de entidad asociado será restaurado y utilizado para más procesamiento.

Por último, los **beans dirigidos por mensajes** se utilizan para programar objetos que deberán reaccionar a mensajes que llegan (y asimismo, ser capaces de enviar mensajes). Los beans dirigidos por mensajes no pueden ser invocados directamente por un cliente, sino que se ajustan a una forma de comunicación de *publicación-suscripción*, la cual se presentó brevemente en el capítulo 4. A lo que se reduce esto es a que un bean dirigido por mensajes es automáticamente invocado por el servidor cuando recibe un mensaje específico *m* al cual el servidor (o más bien una aplicación que está alojando) se había suscrito previamente. El bean contiene un código de aplicación para manejar el mensaje, después de lo cual simplemente lo desecha. Los beans dirigidos por mensajes son vistos, por tanto, como sin estado. En el capítulo 13 volveremos a tocar extensamente este tipo de comunicación.

10.1.3 Ejemplo: Objetos compartidos distribuidos Globe

A continuación examinaremos un tipo completamente diferente de sistema distribuido basado en objetos. **Globe** es un sistema en el cual la escalabilidad desempeña un rol central. Todos los aspectos que tienen que ver con la construcción de un sistema de área amplia a gran escala y capaz de soportar un enorme número de usuarios y objetos motivaron el diseño de Globe. Para este método, la forma en que los objetos son vistos resulta fundamental. Al igual que en otros sistemas basados en objetos, en Globe se espera que los objetos encapsulen estado y operaciones incluidas en el estado.

Una diferencia importante con otros sistemas basados en objetos es que se espera que éstos también encapsulen la implementación de políticas que prescriban la distribución del estado de un objeto a través de varias máquinas. En otros términos, cada objeto determina cómo se distribuirá su estado en sus réplicas. Cada objeto controla también sus propias políticas en otras áreas.

En general, en Globe los objetos son puestos a cargo tanto como sea posible. Por ejemplo, un objeto decide cómo, cuándo y adónde su estado deberá ser migrado. También, un objeto decide si su estado tiene que ser replicado, y de ser así, cómo deberá ocurrir la replicación. Además, un objeto puede determinar su política de seguridad y su implementación. A continuación, describimos cómo lograr el encapsulado.

Modelo de objeto

A diferencia de la mayoría de los sistemas distribuidos basados en objetos, Globe no adopta el modelo de objeto remoto. En vez de eso, en Globe los objetos pueden estar físicamente distribuidos, ello significa que el estado de un objeto puede ser distribuido y replicado a través de múltiples procesos. Esta organización se muestra en la figura 10-3, donde aparece un objeto distribuido a través de cuatro procesos, cada uno ejecutándose en una máquina diferente. En Globe los objetos se denominan **objetos compartidos distribuidos**, para reflejar que normalmente son compartidos por varios procesos. El modelo de objeto se deriva de los objetos distribuidos utilizados en Orca tal como se describe en Bal (1989). Se han seguido métodos similares para objetos fragmentados (Makpangou y cols., 1994).

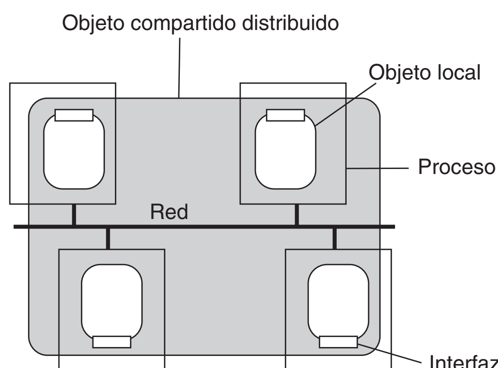


Figura 10-3. Organización de un objeto Globe compartido distribuido.

A un proceso que está ligado a un objeto compartido distribuido se le ofrece la implementación local de las interfaces provistas por dicho objeto. Tal implementación local se llama **representante local** o simplemente **objeto local**. En principio, el que un objeto local tenga o no estado es completamente transparente para el proceso ligado. Todos los detalles de implementación de un objeto se encuentran ocultos detrás de las interfaces ofrecidas a un proceso. Lo único visible por fuera del objeto local son sus métodos.

Los objetos locales Globe vienen en dos sabores. Un **objeto local primitivo** es un objeto local que no contiene cualesquiera otros objetos locales. Por contraste, un **objeto local compuesto** es un objeto compuesto a partir de múltiples objetos locales (posiblemente compuestos). Se utiliza composición para construir un objeto local requerido para implementar objetos compartidos distribuidos. Este objeto local se muestra en la figura 10-4 y consta de por lo menos cuatro subobjetos.

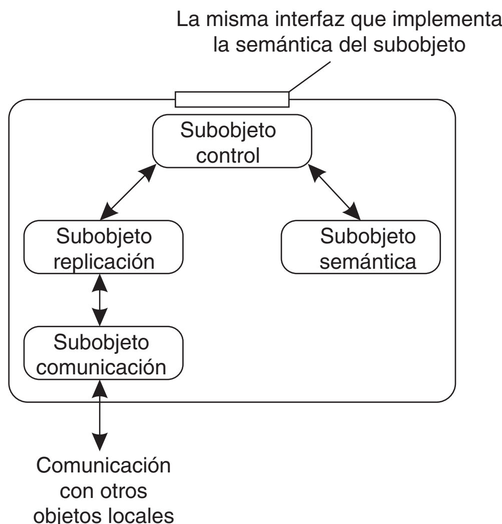


Figura 10-4. Organización general de un objeto local en el caso de objetos compartidos distribuidos en Globe.

El **subobjeto semántica** implementa la funcionalidad provista por un objeto compartido distribuido. En esencia, corresponde a objetos remotos ordinarios, en sabor es similar a los EJB.

El **subobjeto comunicación** se utiliza para proporcionar una interfaz estándar a la red subyacente. Este subobjeto ofrece varios primitivos de traspaso de mensajes para comunicación con y sin conexión. También existen subobjetos de comunicación más avanzados disponibles que implementan interfaces de multitransmisión. Se pueden utilizar subobjetos de comunicación que implementan comunicación confiable, en tanto que otros ofrecen sólo comunicación no confiable.

El **subobjeto replicación** resulta crucial para virtualmente todos los objetos compartidos distribuidos. Este subobjeto implementa la estrategia de distribución para un objeto. Como en el caso del subobjeto comunicación, su interfaz está estandarizada. El subobjeto replicación es responsable de decidir con exactitud cuándo tiene que ser realizado un método provisto por el subobjeto semántica. Por ejemplo, un subobjeto replicación que implementa la replicación activa garantizará que todas las invocaciones a un método sean realizadas en el mismo orden en cada réplica. En este caso, el subobjeto tendrá que comunicarse con los subobjetos replicación ubicados en otros objetos locales que comprenden el objeto compartido distribuido.

El **subobjeto control** se utiliza como intermediario entre las interfaces definidas por el usuario del subobjeto semántica y las interfaces estandarizadas del subobjeto replicación. Además, es responsable de exportar las interfaces del subobjeto semántica al proceso ligado al objeto compartido distribuido. Todas las invocaciones a un método solicitadas por ese proceso son organizadas por el subobjeto control y traspasadas al subobjeto replicación.

El subobjeto replicación finalmente permitirá que el subobjeto control continúe con la solicitud de invocación y devolverá los resultados al proceso. Asimismo, las solicitudes de invocación desde procesos remotos son finalmente pasadas también al subobjeto control. Una solicitud de este tipo es luego desorganizada, tras de lo cual la invocación es realizada por el subobjeto control y regresa los resultados al subobjeto replicación.

10.2 PROCESOS

Los servidores de objetos desempeñan un rol fundamental en sistemas distribuidos basados en objetos, es decir, el servidor diseñado para alojar objetos distribuidos. En lo que sigue, primero abordamos los aspectos generales de los servidores de objetos, y luego analizamos el servidor JBoss de fuente abierta.

10.2.1 Servidores de objetos

Un servidor de objetos es un servidor diseñado para soportar objetos distribuidos. La diferencia importante entre un servidor de objetos general y otros servidores (más tradicionales) es que un servidor de objetos no proporciona, por sí mismo, un servicio específico. Los servicios específicos son implementados por los objetos que residen en el servidor. En esencia, el servidor sólo proporciona los medios necesarios para invocar objetos locales, basado en solicitudes de clientes remotos. En consecuencia, es relativamente fácil cambiar los servicios simplemente con agregar o eliminar objetos.

Un servidor de objetos actúa, por tanto, como un lugar donde residen los objetos. Un objeto se compone de dos partes: datos que representan su estado y el código para ejecutar sus métodos. Que estas partes estén o no separadas, o que las implementaciones del método sean compartidas por múltiples objetos, depende del servidor de objetos. También, existen diferencias en la forma en que cada servidor invoca sus objetos. Por ejemplo, en un servidor de hilos múltiples, a cada objeto se le puede asignar un hilo distinto o se puede utilizar un hilo distinto para cada solicitud de invocación. Enseguida analizamos éstos y otros temas.

Alternativas para invocar objetos

Para que un objeto sea invocado, el servidor necesita saber qué código debe ejecutar, en qué datos debe operar, si debe iniciar un hilo distinto para que se haga cargo de la invocación, y así sucesiva-

mente. Un método simple es asumir que todos los objetos son iguales y que existe sólo una forma de invocar un objeto. Por desgracia, ese método es generalmente inflexible y a menudo limita en forma innecesaria a los desarrolladores de objetos distribuidos.

Un método mejor es que un servidor soporte políticas diferentes. Consideremos, por ejemplo, los objetos transitorios. Recordemos que un objeto transitorio es un objeto que existe sólo en tanto su servidor exista, pero posiblemente durante un lapso más corto. Una copia de un archivo de sólo lectura guardado en la memoria típicamente podría ser implementada como un objeto transitorio. Asimismo, una calculadora también podría ser implementada como un objeto transitorio. Una política razonable es crear un objeto transitorio a la primera solicitud de invocación y destruirlo en cuanto ya ningún cliente esté ligado a tal objeto.

La ventaja de este método es que un objeto transitorio necesitará los recursos de un servidor sólo en tanto el objeto realmente se requiera. La desventaja es que una invocación puede llevarse cierto tiempo para ser completada, porque primero se tiene que crear el objeto. Por consiguiente, una política alternativa es, en ocasiones, crear todos los objetos transitorios en el momento en que el servidor se inicializa, a expensas del consumo de recursos incluso cuando ningún cliente esté utilizando el objeto.

De igual modo, un servidor podría seguir la política de que cada uno de sus objetos sea colocado en un segmento de su memoria. En otros términos, que los objetos ni compartan el código ni compartan los datos. Esa política puede ser necesaria cuando la implementación de un objeto no separa código y datos, o cuando los objetos tienen que separarse por razones de seguridad. En este segundo caso, el servidor tendrá que proporcionar medidas especiales o requerir apoyo del sistema operativo subyacente para garantizar que no se violen los límites del segmento.

El método alternativo es permitir que los objetos por lo menos compartan su código. Por ejemplo, una base de datos que contiene objetos pertenecientes a la misma clase puede ser implementada eficientemente si la implementación de clase es cargada sólo una vez en el servidor. Cuando llega una solicitud de invocación de un objeto, el servidor sólo necesita buscar el estado del objeto en la base de datos y ejecutar el método solicitado.

Asimismo, existen muchas políticas diferentes con respecto a la creación de hilos. La más simple es implementar el servidor con sólo un hilo de control. Alternativamente, puede tener varios hilos, uno por cada uno de sus objetos. Siempre que llega una invocación para un objeto, el servidor la traspasa al hilo responsable de dicho objeto. Si el hilo está ocupado, la solicitud se pone en cola temporalmente.

La ventaja de este método es que los objetos quedan protegidos automáticamente contra acceso concurrente: todas las invocaciones se serializan a través del hilo asociado con el objeto. Así de simple. Desde luego, también es posible utilizar un hilo distinto para cada solicitud de invocación, para lo cual se requiere que los objetos ya deban estar protegidos contra acceso concurrente. Independiente de utilizar un hilo por cada objeto o un hilo por método es la alternativa de si se crean hilos por demanda o si el servidor mantiene un conjunto de hilos. En general, no existe una política única que sea la mejor. Cuál utilizar depende de si están disponibles hilos, de cuánto importa el desempeño, y de factores similares.

Adaptador de objetos

Las decisiones sobre cómo invocar un objeto comúnmente se conocen como **políticas de activación**, para enfatizar que en muchos casos el propio objeto primero tiene que ser llevado al espacio de dirección del servidor (es decir, activado) antes de que en realidad pueda ser invocado. Lo que se requiere entonces es un mecanismo para agrupar objetos por política. Tal mecanismo en ocasiones se denomina **adaptador de objetos** o, alternativamente, **envolvedor de objetos**. Un adaptador de objetos se puede considerar mejor como un software que implementa una política de activación específica. El tema principal, sin embargo, es que los adaptadores de objetos vienen como componentes genéricos para ayudar a los desarrolladores de objetos distribuidos, y sólo tienen que ser configurados para una política específica.

Un adaptador de objetos controla uno o más objetos. Debido a que un servidor debe ser capaz de soportar simultáneamente objetos que requieren políticas de activación diferentes, varios adaptadores de objetos pueden residir en el mismo servidor al mismo tiempo. Cuando se entrega una solicitud de invocación al servidor, primero es despachada al adaptador de objetos apropiado, como se muestra en la figura 10-5.

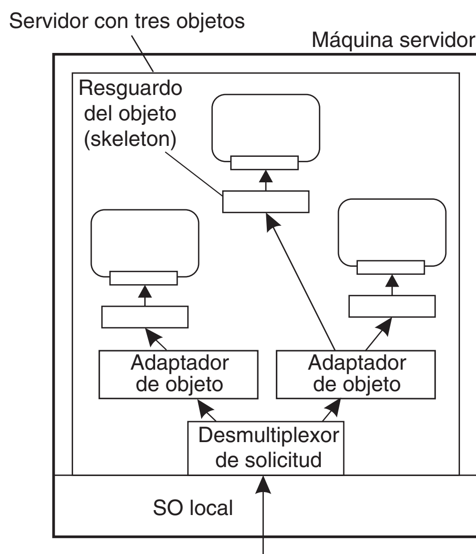


Figura 10-5. Organización de un servidor de objetos que soporta diferentes políticas de activación.

Una observación importante es que los adaptadores de objetos no tienen conocimiento de las interfaces específicas de los objetos que controlan. De lo contrario, nunca podrían ser genéricos. El único tema importante para un adaptador de objetos es que puede extraer una referencia a un objeto de una solicitud de invocación, y posteriormente despachar la solicitud al objeto referido, pero

siguiendo entonces una política de activación específica. Como también se ilustra en la figura 10-5, en lugar de pasar la solicitud directamente al objeto, un adaptador la entrega al resguardo del lado del servidor de dicho objeto. El resguardo, también llamada skeleton, normalmente se genera a partir de las definiciones de interfaz del objeto, desorganiza la solicitud e invoca el método apropiado.

Un adaptador de objetos puede soportar diferentes políticas de activación tan sólo con configurarlo en tiempo de ejecución. Por ejemplo, en sistemas CORBA-compliant (OMG, 2004a) es posible especificar si un objeto debe continuar existiendo después de que su adaptador asociado se ha detenido. Asimismo, un adaptador puede ser configurado para generar identificadores de objeto, o para permitir que la aplicación aporte uno. Como un ejemplo final, un adaptador puede ser configurado para que opere en modo de hilo único y o de hilos múltiples como explicamos antes.

Como un comentario al margen, observe que aunque en la figura 10-5 se ha hablado sobre objetos, nada se ha dicho acerca de lo que en realidad son estos objetos. En particular, se debe recalcar que, como parte de la implementación de tal objeto, el servidor puede acceder (directamente) a bases de datos o llamar rutinas de biblioteca especiales. Los detalles de implementación están ocultos para el adaptador de objetos que se comunica sólo con un skeleton. Como tal, quizá la implementación no tenga nada que hacer con lo que a menudo se ve con objetos a nivel de lenguaje (es decir, tiempo de compilación). Por esta razón, en general, se adopta una terminología diferente. **Sirviente** es el término general utilizado para identificar una sección de código que forma la implementación de un objeto. Desde este punto de vista, un bean Java puede ser visto simplemente como otra clase de sirviente.

10.2.2 Ejemplo: Sistema en tiempo de ejecución Ice

Consideremos cómo se manejan los objetos distribuidos en la práctica. De modo breve, presentamos el sistema de objetos distribuidos Ice, que en parte ha sido desarrollado en respuesta a los embrollos de los sistemas distribuidos basados en objetos comerciales (Henning, 2004). En esta sección, ponemos énfasis en el núcleo de un servidor de objetos Ice y otras partes del sistema se abordan en secciones posteriores.

En Ice, un servidor de objetos no es más que un proceso ordinario que empieza simplemente con la inicialización del sistema en tiempo de ejecución Ice (RTS, del inglés *runtime system*). La base del ambiente en tiempo de ejecución se forma por medio de lo que se llama un *comunicador*. Un comunicador es un componente que maneja varios recursos básicos, de los cuales el más importante está formado por un conjunto de hilos. Asimismo, tendrá una memoria asociada dinámicamente asignada, y así sucesivamente. Además, un comunicador proporciona los medios necesarios para configurar el ambiente. Por ejemplo, es posible especificar longitudes máximas de los mensajes, reintentos de invocación máximos, etcétera.

Un servidor de objetos tendría normalmente sólo un comunicador. Sin embargo, cuando diferentes aplicaciones tienen que estar totalmente separadas y protegidas unas de otras, se puede crear un comunicador aparte (con posiblemente una configuración diferente) dentro del mismo proceso. Cuando menos, tal método separaría los diferentes conjuntos de hilos de modo que si una aplicación los consume todos, esto no afecte a la otra aplicación.

Un comunicador también puede ser utilizado para crear un adaptador de objetos, como se muestra en la figura 10-6. Observemos que el código está simplificado e incompleto. En Henning y Spruiell (2005) se incluyen más ejemplos e información detallada sobre Ice.

```
main(int argc, char* argv[]) {
    Ice::Communicator    ic;
    Ice::ObjectAdaoter   adapter;
    Ice::Object          object;

    ic = Ice::initialize(argc, argv);
    adapter =
        ic->createObjectAdapterWithEnd Points("MyAdapter","tcp -p 10000");
    object = new MyObject;
    adapter->add(object, objetID);
    adapter->activate();
    ic->waitForShutdown();
```

Figura 10-6. Ejemplo de la creación de un servidor de objetos en Ice.

En este ejemplo, empezamos con la creación e inicialización del ambiente en tiempo de ejecución. Una vez hecho esto, se crea un adaptador de objetos. En este caso, se le indica que escuche las conexiones TCP que llegan por el puerto 10000. Veamos que el adaptador se crea en el contexto del comunicador recién creado. Ahora se está en la posición de crear un objeto y agregarlo posteriormente al adaptador. Por último, el adaptador se *activa*, ello significa que, en forma oculta, se activa un hilo que comenzará a escuchar las solicitudes entrantes.

Este código aún no muestra mucha diferenciación en políticas de activación. Las políticas pueden cambiarse modificando las *propiedades* de un adaptador. Una familia de propiedades está dedicada a mantener el conjunto de hilos propio de un adaptador utilizado para manejar las solicitudes entrantes. Por ejemplo, se puede especificar que siempre deberá haber sólo un hilo, serializando efectivamente todos los accesos a objetos que se han agregado al adaptador.

De nueva cuenta, observemos que no se ha especificado MyObject. Como antes, este podría ser un objeto C++ simple, pero también uno que acceda a bases de datos y a otros servicios externos que conjuntamente implementan un objeto. Registrando MyObject con un adaptador, esos detalles de implementación quedan completamente ocultos a los clientes, quienes ahora creen que están invocando un objeto remoto.

En el ejemplo anterior, se crea un objeto como parte de la aplicación, después de lo cual se agrega a un adaptador. Efectivamente, esto significa que tal vez un adaptador tenga que soportar muchos objetos al mismo tiempo, lo cual conduce a un problema potencial de escalabilidad. Una solución alternativa es cargar dinámicamente los objetos en la memoria cuando se requieran. Para

hacer esto, Ice soporta objetos especiales conocidos como *localizadores*. Se llama a un localizador cuando el adaptador recibe una solicitud para un objeto que no ha sido agregado explícitamente. En ese caso, la solicitud se remite al localizador, cuyo trabajo es encargarse de ella.

Para concretar, supongamos que un localizador recibe una solicitud para un objeto del cual sabe que su estado se guarda en un sistema de base de datos relacional. Naturalmente, aquí no hay magia: el localizador ha sido programado explícitamente para manejar ese tipo de solicitudes. En este caso, el identificador del objeto puede corresponder a la clave de un registro donde se guarda el estado. El localizador entonces simplemente consultará la clave, buscará el estado, y así será capaz de procesar la solicitud.

Puede haber más de un localizador agregado a un adaptador. En ese caso, el adaptador podría perder de vista cuáles identificadores de objeto pertenecen al mismo localizador. El uso de múltiples localizadores permite que un solo adaptador soporte muchos objetos. Desde luego, éstos (o más bien su estado) tendrían que ser cargados en tiempo de ejecución, pero este comportamiento dinámico posiblemente haría que el servidor fuera relativamente simple.

10.3 COMUNICACIÓN

Ahora prestemos atención a cómo se maneja la comunicación en sistemas distribuidos basados en objetos. No es de sorprender que estos sistemas, en general, ofrezcan los medios necesarios para que un cliente remoto invoque un objeto. Este mecanismo está basado en gran medida en las llamadas a procedimientos remotos (RPC, por sus siglas en inglés), las cuales se abordaron extensamente en el capítulo 4. Sin embargo, antes de estudiar la comunicación, existen numerosos temas relacionados que debemos conocer.

10.3.1 Vinculación de un cliente a un objeto

Una diferencia interesante entre los sistemas RPC tradicionales que soportan objetos distribuidos es que éstos, en general, proporcionan referencias a objetos a nivel de todo el sistema. Tales referencias a objetos pueden ser pasadas libremente entre los procesos en diferentes máquinas, por ejemplo, como parámetros ante invocaciones a métodos. Ocultando la implementación de la referencia a un objeto, esto es, haciéndola opaca, y quizás incluso utilizándola como la única forma de referirse a objetos, la transparencia de la distribución mejora en comparación con las RPC tradicionales.

Cuando un proceso retiene la referencia a un objeto, primero debe vincular el objeto referido antes de invocar cualquiera de sus métodos. La vinculación hace que se coloque un proxy en el espacio de dirección del proceso, y así se implementa una interfaz que contiene los métodos que el proceso puede invocar. En muchos casos, la vinculación se hace automáticamente. Cuando el sistema subyacente recibe una referencia a un objeto, necesita una forma de localizar el servidor que maneja el objeto real, y coloca un proxy en el espacio de dirección del cliente.

Con **vinculación implícita**, al cliente se le ofrece un mecanismo simple que le permite invocar directamente métodos que utilizan sólo una referencia a un objeto. Por ejemplo, C++ permite

sobrecargar el operador de selección de miembro unario ("→") dejándonos introducir referencias a objeto como si fueran apuntadores ordinarios, tal como se muestra en la figura 10-7(a). Con vinculación implícita, el cliente está ligado transparentemente al objeto en el momento en que la referencia se transforma en el objeto real. Por contraste, con **vinculación explícita**, el cliente primero deberá invocar una función especial para ligarse al objeto antes de que en realidad pueda invocar sus métodos. La vinculación explícita, en general, regresa un apuntador a un proxy que luego llega a estar localmente disponible, como ilustra la figura 10-7(b).

```
Distr_object*obj_ref;        //Declarar una referencia a un objeto a nivel de todo el sistema
obj_ref = …;                 //Inicializar la referencia a un objeto distribuido
obj_ref→do_something( );     //Vincular e invocar implícitamente un método
```

(a)

```
Distr_object obj_ref;        //Declarar una referencia a un objeto a nivel de todo el sistema
Local_object* obj_ptr;       //Declarar un apuntador a los objetos locales
obj_ref = …;                 //Inicializar la referencia a un objeto distribuido
obj_ptr = bind(obj_ref);     //Vincularse y obtener explícitamente un apuntador al proxy local
obj_ptr→do_something( );     //Invocar un método en el proxy local
```

(b)

Figura 10-7. (a) Ejemplo con vinculación implícita utilizando sólo referencias globales. (b) Ejemplo con vinculación explícita utilizando referencias globales y locales.

Implementación de referencias a un objeto

Está claro que la referencia a un objeto debe contener la suficiente información como para permitir que un cliente se vincule a un objeto. Una referencia simple a un objeto incluiría la dirección de red de la máquina donde reside el objeto, junto con un punto final que identifica el servidor que maneja el objeto, más una indicación acerca de qué objeto se trata. Observe que una parte de esta información será aportada por un adaptador de objetos. No obstante, este esquema tiene varias desventajas.

En primer lugar, si la máquina del servidor se congela y al servidor se le asigna un punto de terminación diferente después de recuperarse, todas las referencias a un objeto se vuelven inválidas. Este problema se resuelve como se hizo en DCE: hacer que un centinela local por cada máquina escuche a un punto de terminación bien conocido y que no pierda de vista las asignaciones del servidor al punto de terminación en una tabla de puntos de terminación. Cuando vinculamos un cliente a un objeto, primero preguntamos al centinela por el punto de terminación actual del servidor. Este método requiere se codifique el ID del servidor en la referencia a un objeto para utilizarla como indicador en la tabla de puntos de terminación. Siempre se requiere que el servidor, a su vez, se registre con el centinela local.

Sin embargo, codificar la dirección de la red de la máquina del servidor en la referencia a un objeto no siempre es una buena idea. El problema con este método es que el servidor nunca puede moverse a otra máquina sin invalidar todas las referencias a los objetos que maneja. Una solución

evidente es expandir la idea de un centinela local que mantiene una tabla de puntos de terminación para un **servidor de localización** que no pierde de vista la máquina donde el servidor de un objeto está funcionando actualmente. La referencia a un objeto contendría entonces la dirección de la red del servidor de localización, junto con un identificador del servidor a nivel de todo el sistema. Observemos que esta solución se parece a la implementación de espacios para nombres simples como se vio en el capítulo 5.

Hasta ahora hemos supuesto tácitamente que el cliente y el servidor ya han sido configurados de algún modo para que utilicen el mismo protocolo. Esto no sólo significa que utilizan el mismo protocolo de transporte, por ejemplo, TCP; además significa que utilizan el mismo protocolo de transporte para organizar y desorganizar parámetros. También deben utilizar el mismo protocolo para establecer una conexión inicial, manejar errores y controlar el flujo de igual modo, y así sucesivamente.

Podemos dejar de lado esta suposición siempre que se agregue más información en la referencia a un objeto. Tal información debe incluir la identificación del protocolo utilizado para vincularse a un objeto y de aquellos protocolos soportados por el servidor. Por ejemplo, un solo servidor puede soportar al mismo tiempo datos que llegan a través de una conexión TCP y también datagramas UDP. Es entonces responsabilidad del cliente obtener una implementación proxy para, por lo menos, uno de los protocolos identificados en la referencia a un objeto.

Este método puede incluso llevarse un paso más adelante e incluir un **control de implementación** en la referencia a un objeto, el cual se refiere a una implementación completa de un proxy que el cliente pueda cargar dinámicamente cuando se vincule al objeto. Por ejemplo, un control de implementación podría adoptar la forma de una URL que apunta a un directorio de archivos, tal como *ftp://ftp.clientware.org/proxies/java/proxy-v1a.zip*. El protocolo de vinculación tendría entonces que prescribir que tal archivo deberá ser dinámicamente descargado, desempacado, instalado, y posteriormente instanciado. El beneficio de este método es que el cliente no tiene que preocuparse acerca de si dispone de una implementación de un protocolo específico. Además, da al desarrollador del objeto la libertad de diseñar proxies específicos para objetos. Sin embargo, no tenemos que tomar medidas especiales de seguridad para que el cliente pueda confiar en el código descargado.

10.3.2 Invocaciones a métodos remotos estáticas versus dinámicas

Después de que un cliente se liga a un objeto, puede invocar sus métodos a través del proxy. Tal **invocación de un método remoto**, o simplemente **RMI** (del inglés *remote method invocation*), es muy similar a una RPC cuando se trata de temas tales como la organización y el traspaso de parámetros. Una diferencia esencial entre RMI y RPC es que las RMI, en general, soportan referencias a un objeto a nivel de todo el sistema como ya explicamos. También, no es necesario disponer de resguardos para propósito general del lado del cliente y del lado del servidor. En cambio, se pueden acomodar con facilidad resguardos propios del objeto, como también explicamos antes.

La forma usual de proporcionar soporte RMI es especificando las interfaces del objeto en un lenguaje de definición de interfaz, similar al método seguido con las RPC. De modo alterno, podemos utilizar un lenguaje basado en objetos, tal como Java, que maneje automáticamente la genera-

ción de resguardos. Este método de utilizar definiciones de interfaz predefinidas se conoce, en general, como **invocación estática**. Las invocaciones estáticas requieren que se conozcan las interfaces de un objeto cuando la aplicación del cliente se está desarrollando. También implica que si cambian las interfaces, entonces la aplicación del cliente debe ser recompilada antes de utilizar las interfaces nuevas.

Como una alternativa, las invocaciones a métodos también pueden hacerse de una manera más dinámica. En particular, en ocasiones resulta conveniente ser capaces de *componer* una invocación a un método en tiempo de ejecución, esto es conocido también como **invocación dinámica**. La diferencia esencial con la invocación estática es que una aplicación selecciona en tiempo de ejecución qué método invocará a un objeto remoto. La invocación dinámica adopta, en general, una forma como la siguiente:

```
invoke(object_method, input.parameters, output_parameters);
```

donde *object* identifica el objeto distribuido, *method* es un parámetro que especifica con exactitud qué método deberá ser invocado, *input_parameters* es una estructura de datos que retiene los valores de los parámetros de entrada del método, y *output_parameters* se refiere a una estructura de datos donde los valores de salida pueden ser almacenados.

Por ejemplo, consideremos la anexión de un entero *int* a un objeto de archivo *fobject*, para lo cual el objeto proporciona el método append. En este caso, una invocación estática adoptaría la forma

```
fobject.append(int)
```

en tanto que la invocación dinámica pudiera verse algo así como

```
invoke(fobject, id(append), int)
```

donde la operación *id*(*append*) regresa un identificador para el método append.

Para ilustrar la utilidad de las invocaciones dinámicas, consideremos un objeto navegador utilizado para examinar conjuntos de objetos. Supongamos que el navegador soporta invocaciones a objetos remotos. Tal navegador es capaz de enlazarse a un objeto distribuido para posteriormente presentar la interfaz del objeto a su usuario. A éste se le podría pedir entonces que seleccione un método y proporcione valores para sus parámetros, después de lo cual el navegador puede hacer la invocación. Típicamente, dicho navegador de objetos deberá ser desarrollado para que soporte cualquier interfaz posible. Tal método requiere que las interfaces puedan ser inspeccionadas en tiempo de ejecución, y que las invocaciones a métodos puedan ser construidas dinámicamente.

Otra aplicación de las invocaciones dinámicas es un servicio de procesamiento por lotes al cual puedan ser entregadas solicitudes de invocación junto con el tiempo en que la invocación deberá hacerse. El servicio puede ser implementado por una cola de solicitudes de invocación, ordenadas por el tiempo en que las invocaciones tienen que ser atendidas. El bucle principal del servicio simplemente esperaría hasta que se programe la siguiente invocación, se quite la invocación de la cola, y se llame a invoke en la forma que se mencionó antes.

10.3.3 Paso de parámetros

Como la mayoría de los sistemas RMI soportan referencias a objetos a nivel de todo el sistema, el paso de parámetros en invocaciones a métodos generalmente es menos restrictivo que en el caso de las RPC. Sin embargo, existen algunas sutilezas que pueden hacer que las RMI sean más engañosas de lo que inicialmente se podría esperar, tal como analizamos brevemente en las páginas siguientes.

Primero consideremos la situación en que sólo existen objetos distribuidos. En otros términos, todos los objetos presentes en el sistema pueden ser accesados desde máquinas remotas. En ese caso, podemos utilizar consistentemente las referencias a objetos como parámetros en las invocaciones a métodos. Las referencias son pasadas por valor, y por tanto copiadas de una máquina a otra. Cuando un proceso recibe una referencia a un objeto como resultado de la invocación a un método, simplemente puede enlazarse al objeto para cuando se requiera más adelante.

Por desgracia, utilizar sólo objetos distribuidos puede ser altamente ineficiente, sobre todo cuando los objetos son pequeños, tales como enteros o, peor aún, booleanos. Cada invocación realizada por un cliente que no está colocada en el mismo servidor que el objeto genera una solicitud entre espacios de dirección diferentes o, peor aún, entre diferentes máquinas. Por consiguiente, las referencias a objetos remotos y aquellas a objetos locales a menudo se tratan en forma diferente.

Cuando se invoca un método con la referencia a un objeto como parámetro, dicha referencia se copia y pasa como un parámetro de valor sólo cuando se refiere a un objeto remoto. En este caso, el objeto es literalmente pasado mediante referencia. Sin embargo, cuando la referencia se refiere a un objeto local, es decir un objeto situado en el mismo espacio de dirección que el cliente, el objeto referido se copia en su totalidad y pasa junto con la invocación. En otros términos, el objeto se pasa por valor.

Estas dos situaciones se ilustran en la figura 10-8, la cual muestra un programa cliente ejecutado en la máquina A y un programa servidor en la máquina C. El cliente tiene una referencia a un objeto local $O1$ que utiliza como parámetro cuando invoca al programa servidor en la máquina C. Además, retiene una referencia a un objeto remoto $O2$ que reside en la máquina B, la cual también se utiliza como parámetro. Cuando se invoca el servidor, se pasa una copia de $O1$ al servidor en la máquina C, junto con sólo una copia de la referencia a $O2$.

Observe que ya se trate de una referencia a un objeto local o a un objeto remoto, se puede ser altamente transparente, como en Java. En Java, la distinción es visible sólo porque los objetos locales son esencialmente de un tipo de datos diferente que los objetos remotos. De lo contrario, ambos tipos de referencias se tratan más o menos igual [vea también Wollrath y cols. (1996)]. Por otra parte, cuando se utiliza un lenguaje de programación convencional tal como C, una referencia a un objeto local puede ser tan simple como un apuntador, el cual nunca puede ser utilizado para referirse a un objeto remoto.

El efecto colateral de invocar un método con la referencia a un objeto como parámetro es que podemos estar *copiando* un objeto. Desde luego, ocultar este aspecto es inaceptable, por lo que resulta necesario hacer una distinción explícita entre objetos locales y objetos distribuidos. Claramente, esta distinción no sólo viola la transparencia de distribución, sino que también vuelve más difícil escribir aplicaciones distribuidas.

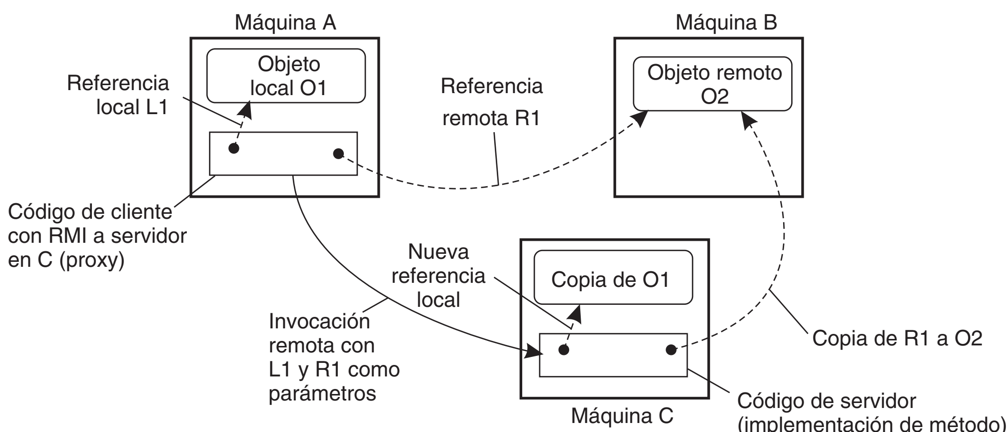


Figura 10-8. Situación en que se transfiere un objeto por referencia o por valor.

10.3.4 Ejemplo: RMI Java

En Java, los objetos distribuidos se incorporaron al lenguaje. Un objetivo importante fue conservar de la semántica de los objetos no distribuidos tanto como fuera posible. En otros términos, los desarrolladores del lenguaje Java apuntaron a un grado más alto de transparencia de distribución. Sin embargo, como se verá, también decidieron hacer aparente la distribución en los casos en que un alto grado de transparencia era simplemente demasiado ineficiente, difícil, o imposible de realizar.

Modelo Java de objetos distribuidos

Java también adopta objetos remotos como la única forma de objetos distribuidos. Recordemos que un objeto remoto es un objeto distribuido cuyo estado siempre reside en una sola máquina, pero cuyas interfaces pueden ponerse a la disposición de procesos remotos. Las interfaces se implementan como siempre por medio de un proxy, el cual ofrece exactamente las mismas interfaces que el objeto remoto. Un proxy aparece como un objeto local en el espacio de dirección del cliente.

Existen sólo algunas, aunque sutiles e importantes, diferencias entre objetos remotos y objetos locales. En primer lugar, la clonación de objetos locales o remotos es diferente. La clonación de un objeto local *O* produce un objeto nuevo del mismo tipo que *O* con exactamente el mismo estado. Esta clonación regresa, por tanto, una copia exacta del objeto clonado. Esta semántica es difícil de aplicar a un objeto remoto. Si fuésemos a hacer una copia exacta de un objeto remoto, no sólo tendríamos que clonar el objeto en su servidor, sino también el proxy en cada cliente que esté ligado en el momento al objeto remoto. La clonación de un objeto remoto es, por consiguiente, una operación que puede ser ejecutada sólo por el servidor. Produce una copia exacta del objeto que se

encuentra en el espacio de dirección del servidor. Por tanto, no se clonan los proxies del objeto. Si un cliente ubicado en una máquina remota desea acceder al objeto clonado en el servidor, primero tendrá que ligarse otra vez al objeto.

Invocación Java a un objeto remoto

Como la distinción entre objetos locales y remotos es difícilmente visible a nivel de lenguaje, Java también puede ocultar la mayor parte de las diferencias durante una invocación a un método remoto. Por ejemplo, cualquier primitivo o tipo de objeto puede ser pasado como parámetro a una RMI, siempre que la organización se aplique sólo al tipo. En terminología Java, esto significa que debe ser **serializable**. Aunque, en principio, la mayoría de los objetos pueden ser serializados, la serialización no siempre está permitida o es posible. Típicamente, objetos que dependen de una plataforma, tales como descriptores de archivo y sockets, no pueden ser serializados.

La única distinción hecha entre objetos locales y remotos durante una RMI es que los objetos locales son pasados por valor (incluidos objetos grandes como las matrices), en tanto que los objetos remotos son pasados por referencia. En otros términos, un objeto local primero se copia, después de lo cual la copia se utiliza como valor de parámetro. Para un objeto remoto, una referencia al objeto se pasa como parámetro en lugar de una copia del objeto, como también se muestra en la figura 10-8.

En RMI Java, una referencia a un objeto remoto se implementa esencialmente como se explica en la sección 10.3.3. Tal referencia se compone de la dirección de red y del punto de terminación del servidor, así como también de un identificador local del objeto en el espacio de dirección del servidor. Ese identificador local es utilizado sólo por el servidor. Como ya explicamos, una referencia a un objeto remoto también necesita codificar el protocolo utilizado por el cliente y el servidor para comunicarse. Para entender cómo se codifica la pila en el caso de RMI Java, es importante darnos cuenta de que en Java cada objeto es una instancia de una clase. Una clase, a su vez, contiene una implementación de una o más interfaces.

En esencia, un objeto remoto se construye a partir de dos clases diferentes. Una clase contiene una implementación del código del lado del servidor, el cual se conoce como *clase de servidor*. Esta clase contiene una implementación de la parte del objeto remoto que se ejecutará en un servidor. En otros términos, contiene la descripción del estado del objeto, así como también una implementación de los métodos que operan en este estado. El resguardo del lado del servidor, es decir, el skeleton, se genera a partir de las especificaciones de interfaz del objeto.

La otra clase contiene una implementación del código del lado del cliente, el cual se llama *clase de cliente*. Esta clase contiene una implementación de un proxy. Al igual que el skeleton, esta clase también se genera a partir de la especificación de la interfaz del objeto. En su forma más simple, lo único que hace un proxy es convertir cada invocación a un método en un mensaje enviado a la implementación del lado del servidor del objeto remoto, y convertir un mensaje de respuesta en el resultado si un método llama. Por cada llamada o invocación, el proxy establece una conexión con el servidor, la cual termina posteriormente cuando finaliza la invocación. Para este propósito, el proxy necesita la dirección de red del servidor y el punto de terminación, como ya se mencionó.

Esta información, junto con el identificador local del objeto en el servidor, siempre se guarda como parte del estado de un proxy.

Por consiguiente, un proxy tiene toda la información que necesita para permitir que un cliente invoque métodos del objeto remoto. En Java, los proxies son serializables. En otros términos, es posible organizar un proxy y enviarlo como una serie de bytes a otro proceso, donde se le puede desempaquetar y utilizar para invocar métodos en relación con el objeto remoto. En otros términos, se puede utilizar un proxy como referencia a un objeto remoto.

Este método es compatible con la forma de Java de integrar objetos locales y distribuidos. Recordemos que en una RMI, un objeto local se pasa copiándolo, en tanto que el objeto remoto se traspasa por medio de una referencia a un objeto a nivel de todo el sistema. Un proxy se trata simplemente como un objeto local. Por consiguiente, en una RMI, es posible pasar un proxy serializable como parámetro. El efecto colateral es que dicho proxy puede ser utilizado como referencia al objeto remoto.

En principio, cuando se organiza un proxy su implementación completa, es decir, todo su estado y código, se convierte en una serie de bytes. La organización de un código como éste no es muy eficiente y puede conducir a referencias muy grandes. Por consiguiente, cuando se organiza un proxy en Java, lo que en realidad sucede es que se genera un control de implementación, especificando con precisión qué clases se requieren para construir un proxy. Posiblemente, algunas de estas clases tienen que ser descargadas primero de un sitio remoto. El control de implementación reemplaza al código que se desorganizó como parte de la referencia a un objeto remoto. Realmente, en Java las referencias a objetos remotos son del orden de unos cuantos cientos de bytes.

Este método de referirse a objetos remotos es altamente flexible y una de las características distintivas de RMI Java (Waldo, 1998). En particular, permite soluciones específicas de objetos. Por ejemplo, consideremos un objeto remoto cuyo estado cambia sólo de vez en cuando. Ese objeto puede ser transformado en un objeto verdaderamente distribuido copiando todo su estado a un cliente en tiempo de vinculación. Cada vez que el cliente invoca un método, opera en la copia local. Para garantizar la consistencia, cada invocación también verifica si cambió el estado en el servidor, en cuyo caso la copia local es actualizada. Asimismo, los métodos que modifican el estado son remitidos al servidor. El desarrollador del objeto remoto tendrá que implementar entonces sólo el código necesario del lado del cliente y descargarlo dinámicamente cuando el cliente se vincule al objeto.

El poder pasar proxies como parámetros funciona sólo porque cada proceso acciona la misma máquina virtual Java. En otros términos, cada proceso funciona en el mismo entorno de ejecución. Un proxy al que se empaqueta simplemente se le desempaqueta en el lado receptor, después de lo cual su código puede ser ejecutado. Por contraste, en DCE por ejemplo, el paso de resguardos es imposible, ya que diferentes procesos pueden funcionar en ambientes de ejecución que difieren con respecto a lenguaje, sistema operativo, y hardware. En cambio, un proceso DCE primero tiene que vincularse (en forma dinámica) en un resguardo localmente disponible que haya sido compilado antes específicamente para el ambiente de ejecución del proceso. Pasando una referencia a un resguardo como parámetro en una RPC, es posible referirse a objetos a través de los límites del proceso.

10.3.5 Manejo de mensajes basado en objetos

Aunque la RMI es la forma preferida de manejar la comunicación en sistemas distribuidos basados en objetos, el manejo de mensajes también ha encontrado su camino como una importante alternativa. Existen varios sistemas de manejo de mensajes basados en objetos, y como puede esperarse, ofrecen casi la misma funcionalidad. En esta sección examinaremos a fondo el manejo de mensajes CORBA, en parte porque también proporciona una interesante forma de combinar la invocación a métodos y la comunicación orientada a mensajes.

CORBA es una especificación muy conocida para sistemas distribuidos. Con el paso de los años han surgido varias implementaciones, aunque está por verse a qué grado llegará CORBA a volverse verdaderamente popular. Sin embargo, aparte de la popularidad, las especificaciones CORBA son amplias (lo cual para muchos también significa que son muy complejas). Reconociendo la popularidad de los sistemas de manejo de mensajes, CORBA actuó rápido para incluir una especificación de un servicio de manejo de mensajes.

Lo que hace diferente al manejo de mensajes en CORBA de otros sistemas es su enfoque inherente basado en objetos para comunicación. En particular, los diseñadores del servicio de manejo de mensajes tuvieron que conservar el modelo de que toda la comunicación ocurre invocando un objeto. En el caso del manejo de mensajes, esta restricción de diseño produjo dos formas de invocaciones de métodos asíncronas (además de otras formas que también fueron provistas por CORBA).

Una **invocación de método asíncrona** es análoga a una RPC asíncrona: el invocador continúa después de iniciar la invocación sin esperar el resultado. En el **modelo de llamada automática** de CORBA, un cliente proporciona un objeto que implementa una interfaz que contiene métodos de llamada automática. Estos métodos pueden ser llamados por el sistema de comunicación subyacente para pasar el resultado de una invocación asíncrona. Un importante tema de diseño es que las invocaciones de método asíncronas no afectan la implementación original de un objeto. En otros términos, es responsabilidad del cliente transformar la invocación síncrona original en una invocación asíncrona: el servidor se presenta con una solicitud de invocación normal (síncrona).

La construcción de una invocación asíncrona se realiza en dos pasos. Primero, la interfaz original tal como la implementa el objeto es reemplazada por dos nuevas interfaces que deben ser implementadas únicamente por el software del lado del cliente. Una interfaz contiene la especificación de los métodos que el cliente puede invocar. Ninguno de estos métodos regresa un valor o tiene algún parámetro de salida. La segunda interfaz es la interfaz de llamada automática. Para cada operación incluida en la interfaz original, contiene un método que será invocado por el sistema en tiempo de ejecución del cliente para pasar los resultados del método asociado invocado por el cliente.

Como un ejemplo, consideremos un objeto que implementa una interfaz simple con sólo un método:

```
int add (in int i, in int j, out int k);
```

Suponga que este método toma dos enteros no negativos i y j y regresa $i + j$ como parámetro de salida k. Se supone que la operación regresa -1 si la operación no se completó con éxito. La transformación de la invocación de método original (síncrona) en una invocación asíncrona con llamadas automáticas se logra generando primero el siguiente par de especificaciones de método

(para nuestro propósito, se eligen nombres convenientes en lugar de seguir las estrictas reglas especificadas en OMG, 2004a):

```
void sendcb_add(in int i, in int j);              //Downcall realizada por el cliente
void replycb_add(in int ret_val, in int k);       //Upcall al cliente
```

En realidad, todos los parámetros de salida de la especificación del método original se eliminan del método al que tiene que llamar el cliente, y regresan como parámetros de entrada de las operaciones de llamada automática. Asimismo, si el método original especificaba un valor de retorno, dicho valor se pasa como parámetro de entrada a la operación de llamada automática.

El segundo paso consiste en compilar las interfaces generadas. En consecuencia, al cliente se le ofrece un resguardo que le permite invocar asincrónicamente sendcb_add. Sin embargo, el cliente necesitará proporcionar una implementación para la interfaz de llamada automática, en nuestro ejemplo conteniendo el método replycb_add. Este último método es invocado por el sistema en tiempo de ejecución local del cliente (RTS, del inglés *runtime system*), y el resultado es una invocación a la aplicación cliente. Observemos que estos cambios no afectan la implementación del lado del servidor del objeto. Utilizando este ejemplo, el modelo de llamada automática se resume en la figura 10-9.

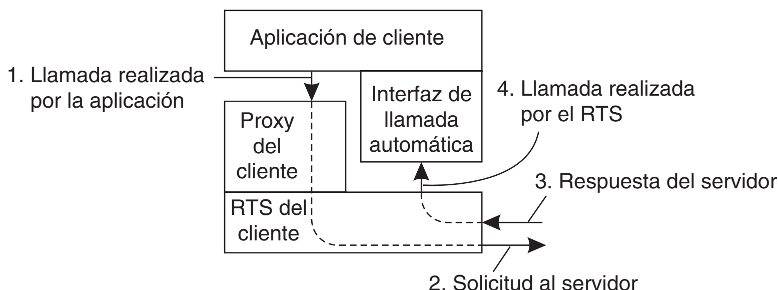


Figura 10-9. Modelo de llamada automática de CORBA de invocación asíncrona a un método.

Como alternativa de las llamadas automáticas, CORBA proporciona un **modelo encuestador**. En éste, al cliente se le ofrece un conjunto de operaciones útiles para encuestar su RTS local en cuanto a los resultados entrantes. Como en el modelo de llamada automática, el cliente es responsable de transformar las invocaciones síncronas originales en invocaciones asíncronas. De nuevo, la mayor parte del trabajo se realiza derivando automáticamente las especificaciones del método apropiado de la interfaz original tal como la implementa el objeto.

Volvamos a nuestro ejemplo. El método add conduce a las siguientes dos especificaciones de método generadas (de nuevo, adoptamos nuestras propias convenciones de asignación de nombres):

```
void sendpoll_add(in int i, in int j);              //Invocado por el cliente
void replypoll_add(out int ret_val, out int k);     //También invocado por el cliente
```

La diferencia más importante entre los métodos encuestadores y de llamada automática es que el método replypoll_add tendrá que ser implementado por el RTS del cliente. Esta implementación puede ser generada automáticamente a partir de las especificaciones de interfaz, de igual modo que se genera el resguardo del lado del cliente como se explicó en el caso de las RPC. El modelo encuestador se resume en la figura 10-10. De nuevo, advierta que la implementación del objeto como aparece en el lado del servidor no tiene que ser cambiada.

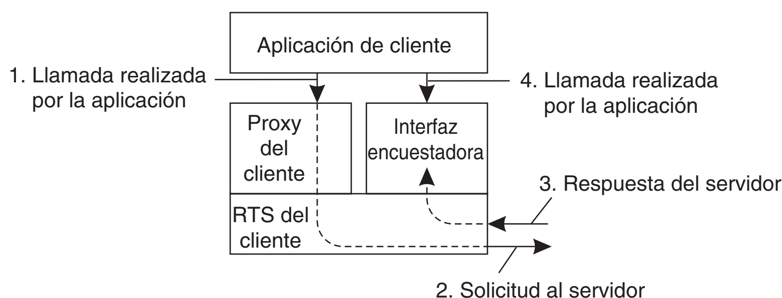


Figura 10-10. Modelo encuestador de CORBA para invocación asíncrona a un método.

Lo que falta señalar en los modelos descritos hasta ahora es que los mensajes enviados entre un cliente y un servidor, incluida la respuesta a una invocación asíncrona, son guardados por el sistema subyacente en el caso de que el cliente o el servidor ya no esté funcionando. Afortunadamente, la mayoría de los temas relacionados con semejante comunicación persistente no afectan el modelo de invocación asíncrona estudiado hasta aquí. Lo que se requiere es reunir un conjunto de servidores para guardar temporalmente mensajes (ya sean solicitudes o respuestas a invocaciones) hasta que puedan ser entregados.

Con esta finalidad, las especificaciones CORBA también incluyen definiciones de interfaz para lo que se conoce como **enrutadores**, que son análogos a los enrutadores de mensajes presentados en el capítulo 4, y los cuales pueden ser implementados, por ejemplo, mediante gestionadores de cola WebSphere de IBM.

Asimismo, Java posee su propio **servicio de gestión de mensajes Java** (**JMS**, del inglés *Java Messaging Service*) que también es similar a lo estudiado aquí con anterioridad [vea Sun Microsystems (2004a)]. Regresaremos a la gestión de mensajes más extensamente en el capítulo 13, cuando analicemos el paradigma de publicación-suscripción.

10.4 ASIGNACIÓN DE NOMBRES

El aspecto interesante de la asignación de nombres en sistemas distribuidos evolucionó en torno a la forma en que son soportadas las referencias a objetos. Ya describimos estas referencias a objetos en el caso de Java, donde efectivamente corresponden a implementaciones de proxy portátiles. Sin embargo, esta es una forma dependiente del lenguaje de ser capaz de referirse a

objetos remotos. De nuevo, tomando a CORBA como ejemplo, vemos cómo la asignación de nombres básica también puede ser provista en una forma independiente del lenguaje y de la plataforma. También presentamos un esquema completamente diferente, el cual se utiliza en el sistema distribuido Globe.

10.4.1 Referencias a objetos CORBA

En CORBA, la forma en que se hace referencia a sus objetos es fundamental. Cuando un cliente retiene una referencia a un objeto, puede invocar los métodos implementados por el objeto referido. Es importante distinguir la referencia a un objeto que un proceso de cliente utiliza para invocar un método, y la referencia implementada por el RTS subyacente.

Un proceso (ya sea cliente o servidor) puede utilizar sólo una implementación dependiente del lenguaje de una referencia a un objeto. En la mayoría de los casos, esta referencia adopta la forma de un apuntador dirigido a la representación local del objeto. Dicha referencia no puede ser pasada del proceso *A* al proceso *B*, ya que tiene significado sólo dentro del espacio de dirección del proceso *A*. En cambio, el proceso *A* primero tiene que organizar al apuntador para convertirlo en una representación independiente del proceso. El RTS proporciona la operación para hacerlo. Una vez transformada la referencia, puede ser traspasada al proceso *B*, el cual puede desempaquetarla otra vez. Observemos que los procesos *A* y *B* pueden ser programas ejecutables escritos en lenguajes diferentes.

Por contraste, el RTS subyacente tendrá su propia representación independiente del lenguaje de una referencia a un objeto. Esta representación incluso puede diferir de la versión empaquetada que entrega a los procesos que desean intercambiar una referencia. Lo importante es que cuando el proceso se refiere a un objeto, su RTS subyacente recibe información suficiente como para que sepa a qué objeto se está haciendo referencia en realidad. Tal información es transferida normalmente por los resguardos del lado del cliente y del servidor generadas a partir de las especificaciones de un objeto.

Uno de los problemas que tenían las primeras versiones de CORBA era que cada implementación podía decidir cómo representar la referencia a un objeto. Por consiguiente, si el proceso *A* deseaba pasar una referencia al proceso *B* como ya se describió, esto en general tendría éxito sólo cuando ambos procesos estuvieran utilizando la misma implementación CORBA. De lo contrario, la versión empaquetada de la referencia retenida por el proceso *A* no significaría nada para el RTS utilizado por el proceso *B*.

Todos los sistemas CORBA actuales soportan la representación independiente del lenguaje de un objeto de referencia, la cual se llama **referencia a objeto interoperable** o **IOR**, por sus siglas en inglés. Si una implementación CORBA utiliza o no IOR internamente no es muy importante. Sin embargo, cuando pasa una referencia a un objeto entre dos sistemas diferentes CORBA, se pasa como IOR. Una IOR contiene toda la información necesaria para identificar un objeto. La disposición general de una IOR se muestra en la figura 10-11, junto con información específica sobre el protocolo de comunicación utilizado en CORBA.

Cada IOR se inicia con un identificador de repositorio. A este identificador se le asigna una interfaz que puede ser guardada y buscada en un repositorio de interfaz. Se utiliza para recuperar

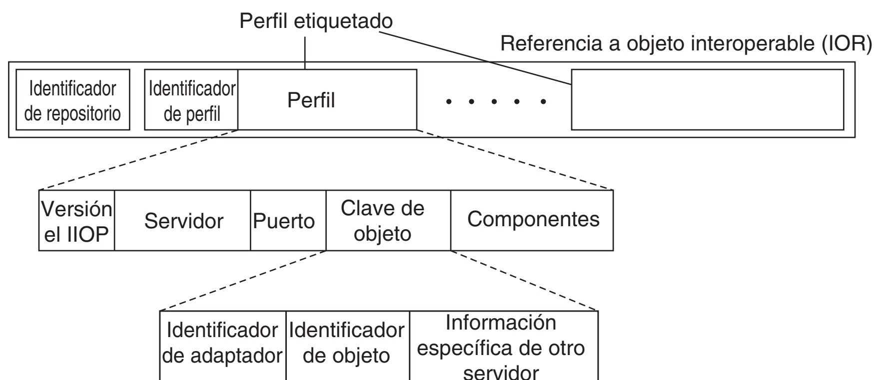


Figura 10-11. Organización de un IOR con información específica para IIOP.

información sobre una interfaz en tiempo de ejecución y puede ayudar en, por ejemplo, la verificación de tipo o en la construcción dinámica de una invocación. Observemos que para que este identificador sea útil, tanto el cliente como el servidor deben tener acceso a la misma interfaz de repositorio, o por lo menos utilizar el mismo identificador para identificar interfaces.

La parte más importante de cada IOR está formada por lo que se llama **perfiles etiquetados**. Cada uno de esos perfiles contiene toda la información necesaria para invocar un objeto. Si el servidor de objetos soporta varios protocolos, se puede incluir información sobre cada protocolo en un perfil etiquetado aparte. CORBA utilizaba el **protocolo Inter-ORB de internet (IIOP**, por sus siglas en inglés) para implementar la comunicación entre nodos. (Un **ORB**, del inglés *Object Request Broker*; es el nombre utilizado por CORBA para su sistema en tiempo de ejecución basado en objetos). IIOP es, en esencia, un protocolo de resguardos para Internet dedicado para invocaciones de métodos remotos soportados para CORBA. En la figura 10-11 también se muestran detalles sobre el perfil utilizado por el IIOP.

El campo *ProfileID* identifica el perfil IIOP en el perfil etiquetado. Su cuerpo se compone de cinco campos. El campo *IIOP version* identifica la versión del IIOP utilizada en este perfil.

El campo *Host* es una cadena que identifica con exactitud en qué servidor está localizado el objeto. El servidor puede ser especificado o por medio de un nombre de dominio DNS completo (tal como *soling.cs.vu.nl*) o con una representación en cadena de su dirección IP, tal como *130.37.24.11*.

El campo *Port* contiene el número de puerto donde el servidor de objetos escucha las peticiones entrantes.

El campo *Object key* contiene información específica del servidor para desmultiplexar las solicitudes entrantes para el objeto apropiado. Por ejemplo, un identificador de objetos generado por un adaptador de objetos CORBA, en general, formará parte de la clave del objeto. Además, esta clave identificará el adaptador específico.

Por último, existe un campo *Components* que de modo alterno contiene más información requerida para invocar apropiadamente el objeto referido. Por ejemplo, este campo puede contener información para indicar cómo debe ser manejada la referencia o qué hacer en caso de que el servidor referido no esté (temporalmente) disponible.

10.4.2 Referencias a objetos Globe

A continuación daremos un vistazo a un forma diferente de hacer referencia a objetos. En Globe, a cada objeto distribuido compartido se le asigna un identificador globalmente único (OID, del inglés *object identifier*), el cual es una cadena de 256 bits. Un OID Globe es un identificador verdadero tal como lo definimos en el capítulo 5. En otros términos, un OID Globe se refiere a, cuando mucho, un objeto compartido distribuido; nunca es reutilizado para otro objeto; y cada objeto tiene cuando mucho un OID.

Los OID Globe pueden ser utilizados sólo para comparar referencias a objetos. Por ejemplo, supongamos que cada uno de los procesos *A* y *B* está ligado a un objeto distribuido compartido. Cada proceso puede solicitar el OID del objeto al que está ligado. Si, y sólo si, dos OID son iguales, entonces se considera que *A* y *B* están ligados al mismo objeto.

A diferencia de las referencias CORBA, los OID Globe no pueden ser utilizados directamente para ponerse en contacto con un objeto. En cambio, para localizar un objeto, es necesario buscar una dirección de contacto del objeto en un servicio de localización. Este servicio regresa una **dirección de contacto**, la cual es comparable a las referencias a objetos dependientes de la ubicación tal como se utilizan en CORBA y otros sistemas distribuidos. Aunque Globe utiliza su propia ubicación de servicio específica, en principio cualquiera de los servicios de localización analizados en el capítulo 5 lo haría.

Ignoremos algunos detalles menores y digamos que una dirección de contacto consta de dos partes. La primera parte es un **identificador de dirección** mediante el cual el servicio de localización puede identificar el nodo apropiado al que deben remitirse las operaciones de insertar o borrar para la dirección de contacto asociada. Recordemos que, como las direcciones de contacto dependen de la localización, es importante insertarlas y borrarlas partiendo de un nodo hoja apropiado.

La segunda parte se compone de información sobre la dirección real, aunque ésta es completamente opaca para el servicio de localización. Para el servicio de localización, una dirección es simplemente un conjunto de bytes que igualmente representa una dirección de red, un apuntador a una interfaz empaquetada, o incluso un proxy completamente empaquetado.

Globe soporta en la actualidad dos clases de direcciones. Una **dirección apilada** representa una suite de protocolo en capas, donde cada capa está representada por el registro de tres campos mostrado en la figura 10-12.

Campo	Descripción
Identificador de protocolo	Constante A que representa un protocolo (conocido)
Dirección de protocolo	Dirección específica del protocolo A
Control de implementación	Referencia a un archivo en un repositorio de clase

Figura 10-12. Representación de una capa de protocolo en una dirección de contacto apilada.

El *identificador de protocolo* es una constante que representa un protocolo conocido. Identificadores de protocolo típicos incluyen *TCP*, *UDP*, e *IP*. El campo *Protocol address* contiene

información específica del protocolo, tal como número de puerto TCP o una dirección de red IPv4. Por último, como alternativa se puede proporcionar un *control de implementación* para indicar dónde se encuentra una implementación preestablecida del protocolo. Típicamente, un control de implementación se representa como una URL.

El segundo tipo de dirección de contacto es una **dirección de instancia**, la cual se compone de los dos campos mostrados en la figura 10-13. De nuevo, la dirección contiene un *control de implementación*, el cual es simplemente una referencia a un archivo incluido en un repositorio de clase donde puede ser encontrada una implementación de un objeto local. Ese objeto local deberá ser cargado por el proceso que actualmente se está ligando al objeto.

Campo	Descripción
Control de implementación	Referencia a un archivo en un repositorio de clase
Cadena de inicialización	Cadena utilizada para inicializar una implementación

Figura 10-13. Representación de una dirección de contacto de instancia.

La carga sigue un protocolo estándar, similar a la carga de clase en Java. Una vez que se carga la implementación y se crea el objeto local, ocurre la inicialización al pasar la *cadena de inicialización* al objeto. En ese momento, el identificador del objeto ha sido completamente resuelto.

Observe la diferencia entre CORBA y Globe en la forma de hacer referencia a un objeto, diferencia que a menudo emerge en sistemas basados en objetos distribuidos. Donde las referencias CORBA contienen información exacta sobre dónde ponerse en contacto con un objeto, las referencias Globe requieren un paso de búsqueda adicional para recuperar dicha información. Esta distinción también aparece en sistemas tales como Ice, donde el equivalente CORBA se conoce como referencia *directa* y el equivalente Globe como referencia *indirecta* (Henning y Spruiell, 2005).

10.5 SINCRONIZACIÓN

Existen sólo algunos temas con respecto a la sincronización en sistemas distribuidos que tienen que ver con la forma de manejar objetos distribuidos. En particular, el hecho de que los detalles de implementación se encuentren ocultos detrás de interfaces puede provocar problemas: cuando un proceso invoca un objeto (remoto), no sabe si dicha invocación conducirá a la invocación de otros objetos. En consecuencia, si un objeto está protegido contra accesos concurrentes, puede ser que tenga un conjunto de candados de los que el proceso invocador no esté al tanto, como se ilustra en la figura 10-14(a).

Por contraste, cuando se trata de recursos de datos tales como archivos o tablas de bases de datos protegidos con candados, el patrón para el control de flujo realmente es invisible para el proceso que utiliza dichos recursos, como se muestra en la figura 10-14(b). Por consiguiente, el proceso también puede ejercer más control en tiempo de ejecución cuando las cosas van mal, tal como renunciar a los candados cuando cree que ha ocurrido un punto muerto. Observe que los sistemas de procesamiento de transacciones siguen generalmente el patrón mostrado en la figura 10-14(b).

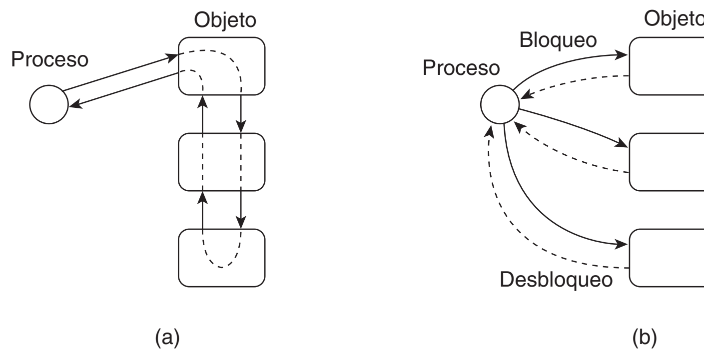


Figura 10-14. Diferencias en el control del flujo para bloquear objetos.

Por tanto, en sistemas distribuidos basados en objetos es importante saber dónde y cuándo ocurre la sincronización. Una ubicación evidente para la sincronización es en el servidor de objetos. Si llegan varias solicitudes de invocación para el mismo objeto, el servidor puede decidir ponerlas en serie (y posiblemente mantener el contacto con un objeto cuando tenga que realizar una invocación remota).

Sin embargo, si se permite al servidor de objetos mantenerse bloqueado las cosas se complican en el caso de que los clientes invocadores se congelen. Por esta razón, el bloqueo también puede hacerse del lado del cliente, un método que ha sido adoptado en Java. Desafortunadamente, este esquema tiene sus propias desventajas.

Como ya se mencionó, en Java la diferencia entre objetos locales y remotos a menudo es difícil de establecer. Las cosas se complican más cuando los objetos están protegidos al declarar que sus métodos están **sincronizados**. Si dos procesos llaman al mismo tiempo a un método sincronizado, sólo un proceso continuará en tanto que el otro será bloqueado. De este modo, podemos garantizar que el acceso a los datos internos de un objeto se serialice por completo. También se puede bloquear un proceso en el interior de un objeto, esperando que alguna condición se vuelva verdadera.

Por lógica, el bloqueo en un objeto remoto es simple. Supongamos que el cliente *A* invoca un método sincronizado de un objeto remoto. Para hacer que el acceso a objetos remotos siempre se vea *exactamente* igual que el acceso a objetos locales, sería necesario bloquear *A* en la plantilla del lado del cliente que implementa la interfaz del objeto y al cual *A* tiene acceso directo. Asimismo, otro cliente ubicado en una máquina distinta también tendría que ser bloqueado localmente antes de que su solicitud pudiera ser enviada al servidor. La consecuencia es que se deben sincronizar diferentes clientes en diferentes máquinas. Como se vio en el capítulo 6, la sincronización distribuida puede ser bastante compleja.

Un método alternativo sería permitir el bloqueo sólo en el servidor. En principio, esto funciona bien, pero surgen problemas cuando un cliente se congela mientras su invocación está siendo manejada por el servidor. Como vimos en el capítulo 8, puede ser que se requieran protocolos relativamente complejos para manejar esta situación, y los cuales pueden afectar significativamente el desempeño total de las invocaciones a métodos remotos.

Por consiguiente, los diseñadores de RMI Java decidieron restringir el bloqueo en objetos remotos sólo a los proxies (Wollrath y cols., 1996). Esto significa que se evitará que los hilos del mismo proceso accedan al mismo tiempo al mismo objeto remoto, pero no se evitará que los hilos de diferentes procesos lo hagan. Desde luego, estas semánticas de sincronización son engañosas: a nivel sintáctico (es decir, cuando se lee el código fuente o de origen), podemos ver un diseño agradable, limpio. Sólo cuando realmente se ejecuta la aplicación distribuida es posible observar un comportamiento no anticipado que debería haberse abordado durante el diseño. Éste es un claro ejemplo en el que buscar la transparencia de distribución *no* es la forma de proceder.

10.6 CONSISTENCIA Y REPLICACIÓN

Muchos sistemas distribuidos basados en objetos siguen un método tradicional por lo que se refiere al manejo de objetos replicados, al tratarlos efectivamente como contenedores de datos con sus propias operaciones especiales. Por consiguiente, cuando se considera cómo se maneja la replicación en sistemas que soportan beans Java o sistemas distribuidos que cumplen con CORBA, en realidad no hay mucho nuevo sobre lo cual informar aparte de lo que ya presentamos en el capítulo 7.

Por esta razón, nos enfocamos en algunos temas particulares relacionados con la consistencia y la replicación que son más profundos en sistemas distribuidos basados en objetos que en otros sistemas. Primero consideraremos la consistencia y luego pasaremos a las invocaciones replicadas.

10.6.1 Consistencia en las entradas

Como se mencionó en el capítulo 7, en el caso de objetos distribuidos la consistencia de los datos se presenta naturalmente en la forma de consistencia en las entradas. Recordemos que, en este caso, el objetivo es agrupar las operaciones realizadas con datos compartidos por medio de variables de sincronización (por ejemplo, en la forma de bloqueos o candados). Como los objetos combinan en forma natural los datos y las operaciones realizadas con ellos, el bloqueo de objetos durante una invocación serializa el acceso y los mantiene consistentes.

Aun cuando la asociación conceptual de un bloqueo con un objeto es simple, no necesariamente proporciona una solución apropiada cuando se replica un objeto. Existen dos temas a resolver para implementar la consistencia en las entradas. El primero es que se requiere una forma de evitar la ejecución concurrente de varias invocaciones en el mismo objeto. En otros términos, cuando se está ejecutando cualquier método de un objeto, ninguno de los otros métodos pueden ejecutarse. Este requerimiento garantiza la serialización del acceso a los datos internos de un objeto. Tan sólo con utilizar mecanismos de bloqueo locales se garantizará esta serialización.

El segundo tema es que en el caso de un objeto replicado, se debe garantizar que los cambios del estado replicado del objeto sean los mismos. En otros términos, debemos asegurarnos de que no ocurran dos invocaciones independientes a un método en diferentes réplicas al mismo tiempo. Este requerimiento implica que se tienen que ordenar las invocaciones de tal suerte que cada una vea las invocaciones en el mismo orden. Este requerimiento se satisface generalmente en una de dos

formas: (1) utilizando un método basado en primario o (2) mediante multitransmisión en orden total para las réplicas.

En muchos casos, el diseño de objetos replicados se realiza diseñando primero un solo objeto, protegiéndolo posiblemente contra el acceso concurrente mediante bloqueo local, y replicándolo posteriormente. Si se tuviera que utilizar un esquema basado en primario, entonces el desarrollador de la aplicación tendría que hacer un esfuerzo adicional para serializar las invocaciones a objetos. Por lo tanto, a menudo es conveniente asumir que el middleware subyacente soporta multitransmisión en orden total, ya que ésta no requeriría cambios en los clientes ni un esfuerzo de programación adicional por parte de los desarrolladores de la aplicación. Naturalmente, el cómo se realiza la multitransmisión totalmente ordenada por el middleware deberá ser transparente. Todo lo que la aplicación puede saber es que su implementación puede utilizar un esquema basado en el primario, pero muy bien podría basarse en relojes de Lamport.

No obstante, incluso si el middleware subyacente proporciona multitransmisión totalmente en orden, se requiere algo más para garantizar la invocación en orden de un objeto. El problema es de granularidad: aunque todas las réplicas de un servidor de objetos pueden recibir solicitudes de invocación en el mismo orden, debemos garantizar que todos los hilos implementados en los servidores también procesen las réplicas en orden correcto. El problema se ilustra en la figura 10-15.

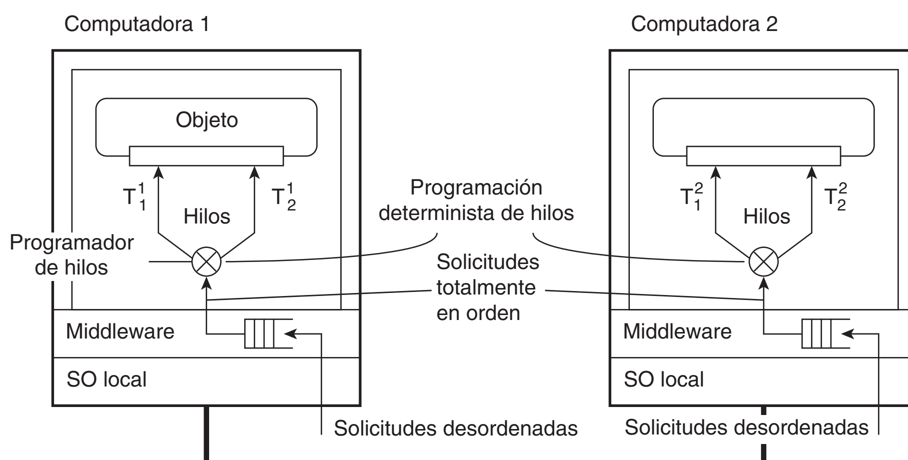


Figura 10-15. Programación determinísta de hilos en servidores de objetos replicados.

Los servidores de varios hilos (de objetos) simplemente captan la solicitud entrante, la transfieren a un hilo disponible, y esperan la llegada de la siguiente solicitud. El programador de hilos del servidor posteriormente asigna la CPU a hilos ejecutables. Naturalmente, si el middleware ha hecho lo mejor que puede para proporcionar un ordenamiento total para la entrega de solicitudes, los programadores de hilos deberán operar de modo determinista para no mezclar en el mismo

objeto el orden de las invocaciones a métodos. En otros términos, si los hilos T_1^1 y T_1^2 de la figura 10-15 manejan la misma solicitud de invocación entrante (replicada), ambas solicitudes deberán ser programadas antes que T_2^1 y T_2^2, respectivamente.

Desde luego, la programación determinista de *todos* los hilos no es necesaria. En principio, si ya tenemos una entrega de solicitudes totalmente ordenada, sólo debemos garantizar que todas las solicitudes para el mismo objeto replicado se manejen en el orden en que fueron entregadas. Tal método permitiría que las invocaciones para diferentes objetos sean procesadas concurrentemente, y sin más restricciones del programador de hilos. Desafortunadamente, existen sólo algunos sistemas que soportan tal concurrencia.

Un método, descrito en Basile y colaboradores (2002), garantiza que los hilos que comparten el mismo bloqueo (local) sean programados en orden similar en cada réplica. En el nivel básico se encuentra un esquema basado en primario en el cual uno de los servidores réplica se adelanta al determinar, para un bloqueo específico, qué hilo va primero. Una mejora que evita la comunicación frecuente entre servidores se describe en Basile y colaboradores (2003). Advierta que los hilos que no comparten un bloqueo pueden operar entonces concurrentemente en cada servidor.

Una desventaja de este esquema es que opera al nivel del sistema operativo subyacente, lo cual significa que cada bloqueo debe ser manejado. Proporcionando información a nivel de la aplicación, el desempeño puede ser mejorado enormemente si se identifican sólo aquellos bloqueos requeridos para señalizar el acceso a objetos replicados (Taiani y cols., 2005). Cuando estudiemos la tolerancia a fallas de Java volveremos a estos temas.

Marcos de trabajo (framework) para replicación

Un aspecto interesante de la mayoría de los sistemas basados en objetos distribuidos es que, por la naturaleza de la tecnología de objetos, a menudo es posible hacer una separación limpia entre la concepción de funcionalidad y el manejo de temas extrafuncionales como la replicación. Tal como explicamos en el capítulo 2, los interceptores forman un poderoso mecanismo para lograr esta separación.

Babaoglu y colaboradores (2004) describen una estructura en la cual utilizan interceptores para replicar beans Java en servidores J2EE. La idea es relativamente simple: las invocaciones a objetos son interceptadas en tres puntos diferentes, como también se muestra en la figura 10-16.

1. Del lado del cliente, justo antes de que la invocación sea transferida al resguardo.
2. En el interior del resguardo del cliente, donde la intercepción forma parte del algoritmo de replicación.
3. Del lado del servidor, justo antes de que el objeto esté a punto de ser invocado.

La primera intercepción se requiere cuando resulta que el invocador está replicado. En ese caso, puede ser que se requiera sincronización con los demás invocadores porque tal vez se trate de una invocación replicada como se explicó antes.

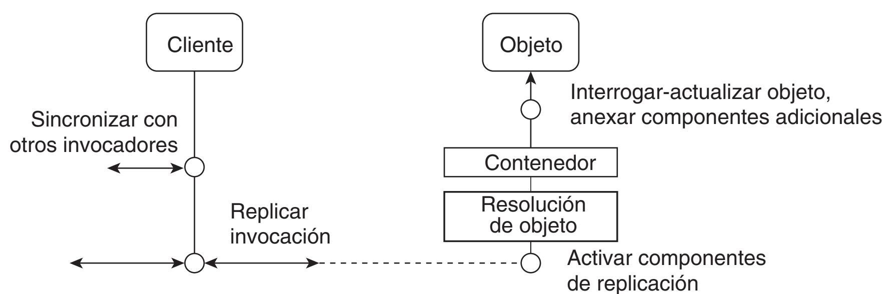


Figura 10-16. Estructura general para separar algoritmos de replicación de objetos en un entorno EJB.

Una vez decidido que la invocación puede ser realizada, el interceptor ubicado en la plantilla del lado del cliente puede tomar decisiones sobre adónde remitir la solicitud, o posiblemente implementar un mecanismo por si ocurre una falla cuando una réplica no puede ser alcanzada.

Por último, el interceptor del lado del cliente maneja la invocación. En realidad, esta intercepción se divide en dos. En el primer punto, un poco después de la llegada de una solicitud y antes de que sea transferida a un adaptador, el algoritmo de replicación asume el control. Puede analizar entonces para quién es la solicitud para permitir la activación, si es necesario, de cualesquiera objetos de replicación requeridos para realizar la replicación. El segundo punto se ubica justo antes de la invocación, permitiendo el algoritmo de replicación para, por ejemplo, obtener y establecer valores de atributo del objeto replicado.

El aspecto interesante es que la estructura puede ser establecida independientemente de cualquier algoritmo de replicación, lo cual conduce entonces a una separación completa de la funcionalidad y la replicación de objetos.

10.6.2 Invocaciones replicadas

Otro problema a resolver es el de las invocaciones replicadas. Consideremos un objeto *A* que llama a otro objeto *B* como se muestra en la figura 10-17. Se supone que el objeto *B* llama además a otro objeto *C*. Si *B* está replicado, cada réplica de *B*, en principio, llamará a *C* en forma independiente. El problema es que *C* es llamado varias veces en lugar de sólo una vez. Si el método llamado en *C* origina la transferencia de $100 000 entonces, claramente, alguien se va a quejar tarde o temprano.

No existen muchas soluciones de propósito general para resolver el problema de invocaciones replicadas. Una solución simple es prohibirlas (Maassen y cols., 2001), lo cual tiene sentido cuando el desempeño está en juego. Sin embargo, cuando se replica en busca de tolerancia a las fallas, la siguiente solución propuesta por Mazouni y colaboradores (1995) puede ser puesta en práctica. Su solución es independiente de la política de replicación, es decir, los detalles exactos de cómo las réplicas se mantienen consistentes. La esencia es proporcionar una capa de comunicación que esté enterada de la réplica sobre la cual se ejecutan los objetos (replicados). Cuando un objeto replicado

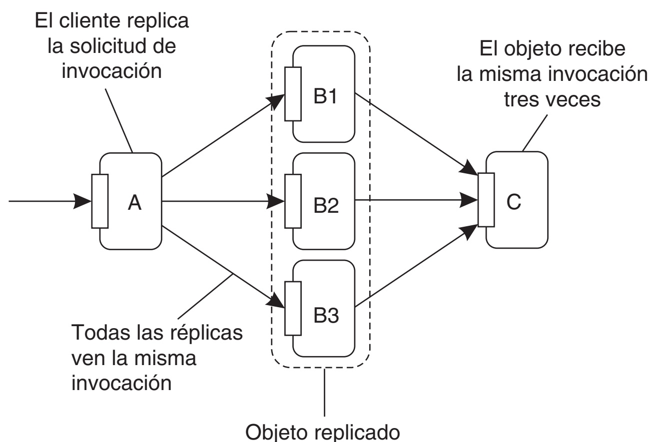


Figura 10-17. Problema de invocaciones a un método replicadas.

B invoca a otro objeto replicado *C*, la solicitud de invocación primero es asignada al mismo y único identificador mediante cada réplica *B*. En ese punto, un coordinador de las réplicas de *B* remite su solicitud a todas las réplicas del objeto *C*, mientras que las demás réplicas de *B* retienen su copia de la solicitud de invocación, como se muestra en la figura 10-18(a). El resultado es que sólo una respuesta es remitida a cada réplica de *C*.

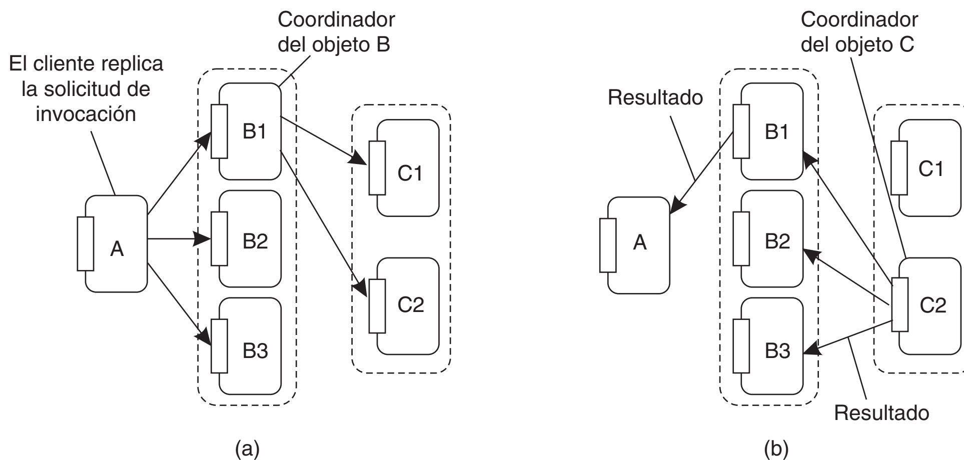


Figura 10-18. (a) Envío de una solicitud de invocación desde un objeto replicado hasta otro objeto replicado. (b) Devolución de una respuesta desde un objeto replicado hasta otro.

Se utiliza el mismo mecanismo para garantizar que sólo un mensaje de respuesta sea regresado a las réplicas de *B*. Esta situación se muestra en la figura 10-18(b). Un coordinador de las réplicas

de *C* advierte que se trata de un mensaje de respuesta replicado que ha sido generado por cada réplica de *C*. Sin embargo, sólo el coordinador remite la respuesta a las réplicas del objeto *B*, en tanto que las demás réplicas de *C* retienen su copia del mensaje de respuesta.

Cuando una réplica de *B* recibe un mensaje de respuesta de una solicitud de invocación que o había remitido a *C* o retenido porque no era el coordinador, la respuesta es transferida entonces al objeto.

En esencia, el esquema que se acaba de describir está basado en la comunicación mediante multitransmisión, pero en prevención de que el mismo mensaje sea multitransmitido por diferentes réplicas. Como tal, es fundamentalmente un esquema basado en el remitente. Una solución alternativa es permitir que una réplica destinataria detecte múltiples copias de mensajes entrantes que pertenezcan a la misma invocación, y que transfiera sólo una copia a su objeto asociado. Los detalles de este esquema se dejan como ejercicio para el lector.

10.7 TOLERANCIA A FALLAS

Al igual que la replicación, en la mayoría de los sistemas distribuidos basados en objetos la tolerancia a fallas utiliza el mismo mecanismo que en otros sistemas distribuidos, y sigue los principios que analizamos en el capítulo 8. Sin embargo, cuando se trata de estandarización, CORBA proporciona indiscutiblemente la especificación más completa.

10.7.1 Ejemplo: CORBA tolerante a fallas

En CORBA, el método básico para ocuparse de las fallas es replicar objetos en la forma de **grupos de objetos**. Dichos grupos consisten en una o más copias idénticas del mismo objeto. Sin embargo, un grupo de objetos puede ser aludido como si fuera un solo objeto. Un grupo ofrece la misma interfaz que las réplicas que contiene. En otros términos, la replicación es transparente para los clientes. Diferentes estrategias de replicación son soportadas, incluidas la estrategia de respaldo primaria, la replicación activa, y la replicación basada en quórum. Todas estas estrategias se presentaron en el capítulo 7. Existen algunas otras propiedades asociadas con grupos de objetos, cuyos detalles se encuentran en OMG (2004a).

Para que proporcionen replicación y transparencia ante las fallas tanto como sea posible, los grupos de objetos no deberán ser distinguibles de los objetos CORBA normales, a menos que una aplicación prefiera lo contrario. Un tema importante, a este respecto, es cómo se hace referencia a los grupos de objetos. El método seguido es utilizar una clase especial de IOR llamada **referencia a grupos de objetos interoperables** (**IOGR**, por sus siglas en inglés). La diferencia clave con una IOR normal es que una IOGR contiene múltiples referencias a *diferentes* objetos, principalmente réplicas presentes en el mismo grupo de objetos. Por contraste, una IOR también puede contener múltiples referencias, pero todas se referirán al *mismo* objeto, aunque tal vez mediante un protocolo de acceso diferente.

Siempre que un cliente transfiere una IOGR a su sistema en tiempo de ejecución (RTS), éste intenta vincularse a una de las réplicas referidas. En el caso de un IIOP, posiblemente el RTS puede utilizar la información adicional que encuentre en uno de los perfiles de IIOP de la IOGR. Tal información puede ser almacenada en el campo *Components* previamente descrito. Por ejemplo, un perfil de IIOP específico puede referirse al primario o a un respaldo de un grupo de objetos, como se muestra en la figura 10-19, por medio de señalizaciones o etiquetas distintas *TAG_PRIMARY* y *TAG_BACKPUP*, respectivamente.

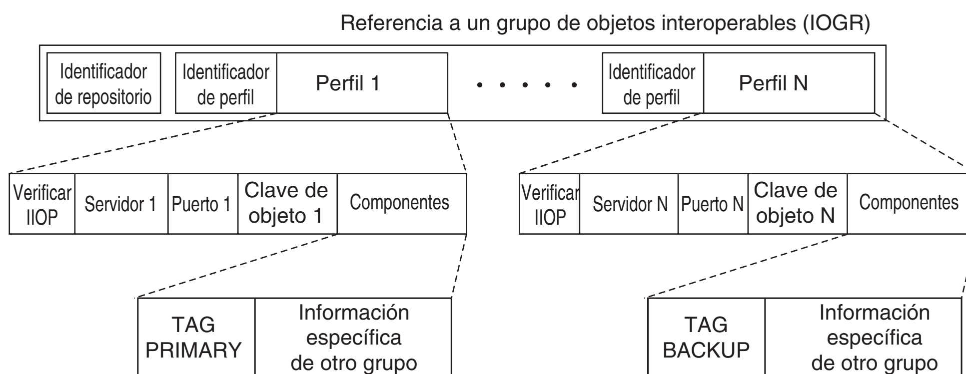


Figura 10-19. Posible organización de un IOGR de un grupo de objetos que tiene un primario y respaldos.

Si falla la vinculación a una de las réplicas, el RTS cliente puede continuar intentando vincularse a otra réplica, siguiendo de tal modo cualquier política para seleccionar a continuación la réplica que mejor se le adapte. El procedimiento de vinculación al cliente es completamente transparente, como si se estuviera vinculando a un objeto CORBA regular.

Un ejemplo de arquitectura

Para soportar grupos de objetos y manejar más gestión de fallas, es necesario agregarle componentes a CORBA. Una posible arquitectura acerca de una versión tolerante a las fallas de CORBA se muestra en la figura 10-20. Esta arquitectura se deriva del sistema Eternal (Moser y cols., 1998; y Narasimhan y cols., 2000), el cual proporciona una infraestructura de tolerancia a fallas construida encima del confiable sistema de comunicación de grupos Tótem (Moser y cols., 1996).

Existen varios componentes que desempeñan un rol importante en esta arquitectura. Con mucho, la más importante es el **gestor de replicación**, el cual es responsable de crear y gestionar un grupo de objetos replicados. En principio, existe sólo un gestor de replicación, aunque puede ser replicado para la tolerancia a fallas.

Como ya hemos expresado, para un cliente no existe ninguna diferencia fundamental entre un grupo de objetos y cualquier otro tipo de objeto CORBA. Para crear un grupo de objetos, un cliente

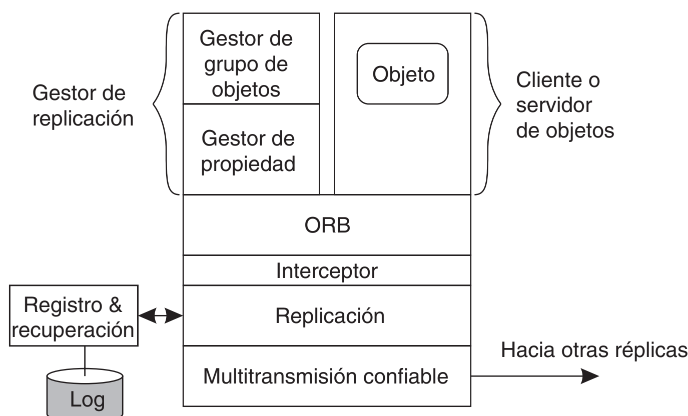


Figura 10-20. Arquitectura ejemplo de un sistema CORBA tolerante a fallas.

simplemente invoca la operación normal create_object tal como es ofrecida, en este caso por el gestor de replicación, especificando el tipo de objeto que se va a crear. El cliente permanece ajeno al hecho de que implícitamente está creando un grupo de objetos. El valor preestablecido dependiente del sistema determina por lo general el número de réplicas creadas cuando se inicia un nuevo grupo de objetos. El gestor de réplicas también es responsable de reemplazar una réplica en el caso de una falla, con lo cual se garantiza que el número de réplicas no se reduzca por debajo de un mínimo especificado.

La arquitectura también muestra el uso de interceptores a nivel del mensaje. En el caso del sistema Eternal, cada invocación es interceptada y transferida a un componente de replicación distinto que mantiene la consistencia requerida para un grupo de objetos y el cual garantiza que los mensajes serán registrados para permitir su recuperación.

Luego se envían invocaciones a los demás miembros del grupo mediante una multitransmisión confiable totalmente ordenada. En el caso de replicación activa, se transfiere una solicitud de invocación a cada objeto replicado entregándola al sistema en tiempo de ejecución subyacente de dicho objeto. Sin embargo, en el caso de replicación pasiva, se transfiere una solicitud de invocación sólo al RTS del primario, en tanto que los demás servidores sólo registran la solicitud de invocación para propósitos de recuperación. Cuando el primario completa la invocación, su estado es multitransmitido a los respaldos.

Esta arquitectura está basada en el uso de interceptores. También existen soluciones alternativas, incluidas aquellas en las cuales la tolerancia a fallas ha sido incorporada al sistema en tiempo de ejecución (que potencialmente afectan la interoperabilidad), o en las cuales se utilizan servicios especiales encima del RTS para proporcionar tolerancia a fallas. Además de estas diferencias, la práctica muestra que existen otros problemas (aún) no cubiertos por el estándar CORBA. Como ejemplo de un problema que ocurre en la práctica, si se crean réplicas en diferentes implementaciones, no existe garantía de que este método funcionará realmente. Una revisión de los diferentes métodos y una evolución de la tolerancia a fallas en CORBA se analizan en Felber y Narasimhan (2004).

10.7.2 Ejemplo: Java tolerante a fallas

Al considerar la popularidad de Java como lenguaje y plataforma útiles para desarrollar aplicaciones distribuidas, también se han realizado esfuerzos para agregar tolerancia a fallas en el sistema en tiempo de ejecución Java. Un método interesante es garantizar que la máquina virtual Java pueda ser utilizada para implementar replicación activa.

En esencia, la replicación activa dicta que los servidores de réplicas funcionen como máquinas determinístas de estado finito (Schneider, 1990). En Java, un excelente candidato para cumplir este rol es la máquina virtual Java (JVM, por sus siglas en inglés). Desafortunadamente, la JVM no es determinista en absoluto. Existen varias causas de su comportamiento no determinista, las cuales fueron identificadas en forma independiente por Napper y colaboradores (2003), y Friedman y Kama (2003):

1. JVM puede ejecutar un código nativo, es decir, un código externo a JVM y provisto a ésta por una interfaz. La JVM trata al código nativo como una caja negra: ve solamente la interfaz, aunque tiene una clave sobre el comportamiento (potencialmente determinísta) que provoca una llamada. Por consiguiente, para utilizar la JVM y efectuar una replicación activa, es necesario asegurarse de que el código nativo se comporte de una forma determinísta.

2. Los datos de entrada pueden no estar sujetos a determinismo. Por ejemplo, una variable compartida que puede ser manipulada usando múltiples hilos puede cambiar para diferentes instancias de la JVM en tanto se permita que los hilos operen concurrentemente. Para controlar este comportamiento, los datos compartidos deberán estar protegidos por lo menos mediante bloqueos o candados. Como resultó, el ambiente en tiempo de ejecución de Java no siempre se adhirió a esta regla, a pesar de su soporte de múltiples hilos.

3. En presencia de fallas, diferentes JVM producirán diferentes resultados que revelan que las máquinas han sido replicadas. Esta diferencia puede provocar problemas cuando las JVM tienen que ser restauradas al mismo estado. Las cosas se simplifican si se puede asumir que todos los resultados son idempotentes (es decir, que simplemente pueden ser repasados), o que son verificables de manera que se pueda comprobar si se produjeron o no antes de una congelación. Observemos que esta suposición no es necesaria para permitir que un servidor de réplicas decida si deberá reejecutar o no una operación.

La práctica muestra que la conversión de la JVM en una máquina de estado finito determinísta de ningún modo es trivial. Un problema a ser resuelto es el hecho de que los servidores de réplicas pueden congelarse. Una posible organización es permitir que los servidores funcionen de acuerdo con un esquema de respaldo primario. En semejante esquema, un servidor coordina todas las acciones que deben ser realizadas, y de vez en cuando instruye al respaldo para que haga lo mismo. Desde luego, se requiere de una cuidadosa coordinación entre el primario y el respaldo.

Observemos que a pesar de que los servidores de réplicas están organizados en un entorno de respaldo primario, la replicación sigue siendo activa: las réplicas se mantienen actualizadas al permitir que cada una ejecute las mismas operaciones en el mismo orden. Sin embargo, para garantizar el mismo comportamiento determinista de todos los demás servidores, el comportamiento de uno se toma como el obligatorio a seguir.

En este entorno, el método seguido por Friedman y Kama (2003) es no permitir que el primario ejecute primero las instrucciones de lo que se llama un **marco**. Un marco consiste en la ejecución de varios cambios de contexto, y termina o porque todos los hilos se están bloqueando para completar la operación de E/S o después de ocurrido un número predefinido de cambios de contexto. Siempre que un hilo emite una operación de E/S, la JVM lo bloquea y lo pone en espera. Cuando se inicia un marco, el primario permite que todas las solicitudes de E/S prosigan, una después de la otra, y los resultados son enviados a las otras réplicas. De este modo, por lo menos se cumple el comportamiento determinista con respecto a las operaciones de E/S.

El problema con este esquema es fácil de advertir: el primario siempre va adelante de las demás réplicas. Existen dos situaciones a considerar. Primero, si un servidor de réplicas diferente del primario se congela, no ocurre ningún daño real excepto que el grado de tolerancia a fallas se reduce. Por otra parte, cuando el primario se congela, es posible presentar la situación en que los datos se pierden (o más bien, las operaciones).

Para reducir al mínimo el daño, el primario funciona cuadro por cuadro. Es decir, envía información actualizada a las demás réplicas sólo después de completar su cuadro en curso. El efecto de este método es que cuando el primario esté funcionando en el marco *k*-ésimo, las demás réplicas tengan toda la información requerida para procesar el marco que precede al *k*-ésimo. El daño puede limitarse a la reducción del tamaño de los marcos al precio de más comunicación entre el primario y los respaldos.

10.8 SEGURIDAD

Desde luego, la seguridad desempeña un rol importante en cualquier sistema distribuido y los sistemas basados en objetos no son la excepción. Cuando se consideran más sistemas distribuidos basados en objetos, el hecho de que los objetos distribuidos sean objetos remotos inmediatamente conduce a una situación en la que las arquitecturas de seguridad para sistemas distribuidos son muy similares. En esencia, cada objeto está protegido mediante autentificación estándar y mecanismos de autorización, como los que estudiamos en el capítulo 9.

Para aclarar cómo puede encajar específicamente la seguridad en un sistema distribuido basado en objetos, examinaremos la arquitectura de seguridad del sistema Globe. Como ya se mencionó, Globe soporta objetos verdaderamente distribuidos en los cuales el estado de un solo objeto puede ser esparcido y replicado a través de múltiples máquinas. Los objetos remotos son tan sólo un caso especial de objetos Globe. Por consiguiente, considerando la arquitectura de seguridad Globe, también podemos ver cómo puede ser igualmente aplicada a sistemas distribuidos basados en objetos más tradicionales. Después de analizar el sistema Globe, examinaremos brevemente la seguridad en sistemas basados en objetos tradicionales.

10.8.1 Ejemplo: Globe

Como ya expusimos, Globe es uno de los pocos sistemas basados en objetos distribuidos donde el estado de un objeto puede estar físicamente distribuido y replicado a través de múltiples máquinas. Este método también presenta problemas de seguridad específicos, los cuales condujeron a una arquitectura como la descrita en Popescu y colaboradores (2002).

Generalidades

Cuando se considera el caso general de invocación de un método aplicado a un objeto remoto, existen por lo menos dos temas importantes desde la perspectiva de seguridad: (1) el que llama está invocando el objeto correcto y (2) el que llama tiene permiso de invocar dicho método. Se hace referencia a estos dos temas como **vinculación a un objeto seguro e invocación a un método seguro**, respectivamente. El primero tiene mucho que ver con la autentificación, en tanto que el segundo implica autorización. Para Globe y otros sistemas que soportan o la replicación o el movimiento de objetos de un lado a otro, se presenta un problema adicional, el llamado **seguridad de plataforma**. Esta clase de seguridad comprende dos temas. En primer lugar, cómo se puede proteger la plataforma donde el objeto (local) está copiado contra cualquier código malicioso contenido en el objeto, y segundo, cómo se puede proteger el objeto contra un servidor de réplicas malicioso.

El ser capaz de copiar objetos en otros servidores también presenta otro problema. Como el servidor de objetos que aloja una copia de un objeto no siempre necesita ser totalmente confiable, debe existir un mecanismo para evitar que todo servidor de réplicas que esté alojando un objeto impida que también ejecute cualesquiera métodos del objeto. Por ejemplo, puede ser que el propietario de un objeto desee limitar la ejecución de métodos de actualización a un pequeño grupo de servidores de réplicas, en tanto que los métodos que sólo leen el estado de un objeto pueden ser ejecutados por cualquier servidor autentificado. El hacer que se cumplan tales políticas puede lograrse mediante el **control de acceso inverso**, el cual analizamos más a fondo a continuación.

Existen varios mecanismos desplegados en Globe para establecer la seguridad. Primero, cada objeto Globe tiene asociado un par de claves pública y privada conocidas como **clave de objeto**. La idea básica es que cualquiera que conozca la clave privada de un objeto puede establecer políticas de acceso para usuarios y servidores. Además, cada réplica tiene una **clave de réplica** asociada, la cual también se construye como un par de claves privada y pública. El servidor de objetos que actualmente está alojando la réplica específica genera este par de claves. Como se verá, la clave de réplica se utiliza para asegurarse de que una réplica específica forme parte de un objeto compartido distribuido dado. Por último, también se asume que cada usuario dispone de un par de claves pública y privada conocido como **clave de usuario**.

Estas claves se utilizan para establecer los diversos derechos de acceso en la forma de certificados. Los certificados son entregados por cada objeto. Existen tres tipos de certificado, como se muestra en la figura 10-21. Un **certificado de usuario** está asociado con un usuario específico y especifica con exactitud qué métodos puede invocar dicho usuario. Con este fin, el certificado contiene una cadena de bits U de la misma longitud que el número de métodos disponibles para el

objeto. $U[i] = 1$ si, y sólo si, al usuario se le permite invocar el método M_i. Asimismo, también existe un **certificado de réplica** que especifica, para un servidor de réplicas dado, qué métodos puede ejecutar. También tiene una cadena de bits asociada R, donde $R[i] = 1$ si, y sólo si, se permite que el servidor ejecute el método M_i.

Certificado de usuario

K^+_{Alicia}
U:0010011100
firma(O, {U, K^+_{Alicia}})

(a)

Certificado de réplica

K^+_{Repl}
R:1100011100
firma(O, {R, K^+_{Repl}})

(b)

Certificado administrativo

K^+_{Adm}
R:1101111100
U:0110011111
D:1
firma(O, {R,U,D,K^+_{Adm}})

(c)

Figura 10-21. Certificados en Globe: (a) certificado de usuario, (b) certificado de réplica, (c) certificado administrativo.

Por ejemplo, el certificado de usuario ilustrado en la figura 10-21(a) dice que Alicia (quien puede ser identificada por su clave pública K^+_{Alicia}), tiene el derecho de invocar los métodos M_2, M_5, M_6, y M_7 (observe que inicia indexando U en 0). Asimismo, el certificado de réplica establece que el servidor que posee K^+_{Repl} puede ejecutar los métodos M_0, M_1, M_5, M_6 y M_7.

Cualquier entidad autorizada puede utilizar un certificado administrativo para emitir certificados de usuario y de réplica. En este caso, las cadenas de bits R y U especifican para qué métodos y entidades se puede crear un certificado. Además, existe un bit que indica si una entidad administrativa puede delegar (una parte de) sus derechos a alguien más. Observemos que cuando Bob, en su rol de administrador, crea un certificado de usuario para Alicia, lo firmará con su propia firma, no con la del objeto. Por consiguiente, el certificado de Alicia tendrá que ser rastreado hasta el certificado administrativo de Bob y, eventualmente, hasta un certificado administrativo firmado con la clave privada del objeto.

Los certificados administrativos sirven bien cuando se considera que algunos objetos Globe pueden ser replicados masivamente. Por ejemplo, puede ser que el propietario de un objeto desee manejar sólo un conjunto relativamente pequeño de réplicas permanentes, pero delegar la creación de réplicas iniciadas por el servidor a los servidores que alojan las réplicas permanentes. En ese caso, el propietario puede decidir permitir que una réplica permanente instale otras réplicas para dar acceso de sólo lectura a todos los usuarios. Siempre que Alicia desee invocar un método de sólo lectura, podrá hacerlo (cuando esté autorizada). Sin embargo, cuando desee invocar un método actualizado tendrá que ponerse en contacto con una de las réplicas permanentes, pues no está permitido que ninguno de los demás servidores de réplicas ejecuten dichos métodos.

Como ya explicamos, en Globe el proceso de vinculación requiere que un identificador de objeto (OID) se transforme en una dirección de contacto. En principio, cualquier sistema que

soporte nombres simples puede ser utilizado para este propósito. Para asociar con seguridad una clave pública de un objeto a su OID, éste se calcula simplemente como una dispersión (hash) seguro de 160 bits de la clave pública. De este modo, cualquiera puede verificar si una clave pública dada pertenece a un OID específico. Estos identificadores también se conocen como **nombres de auto-certificación**, un concepto explorado en el Sistema de archivos seguros (Mazieres y cols., 1999), y los estudiaremos en el capítulo 11.

También podemos verificar si una réplica pertenece a un objeto *O*. En ese caso, simplemente se tiene que inspeccionar el certificado de réplica para *R* y verificar quién lo emitió. El signatario (firmante) puede ser una entidad con derechos administrativos, en cuyo caso se tiene que inspeccionar el certificado administrativo. Lo esencial es que se puede construir una cadena de certificados de los cuales el último está firmado con la clave privada del objeto. En ese caso, se sabe que *R* es una parte de *O*.

Para proteger mutuamente los objetos y servidores uno de otro se despliegan técnicas de código móvil, tal como describimos en el capítulo 9. Con técnicas especiales de auditoría es posible detectar si dichos objetos fueron manipulados, tales técnicas se describirán en el capítulo 12.

Invocación a un método seguro

A continuación examinaremos los detalles de la invocación con seguridad a un método de un objeto Globe. La ruta completa desde la solicitud de una invocación hasta la ejecución de la operación en una réplica se ilustra en la figura 10-22. Si tienen que ejecutar en secuencia 13 pasos en total, como se muestra en la figura y describimos en el texto siguiente.

1. En primer lugar, una aplicación emite una solicitud de invocación haciendo una lla-

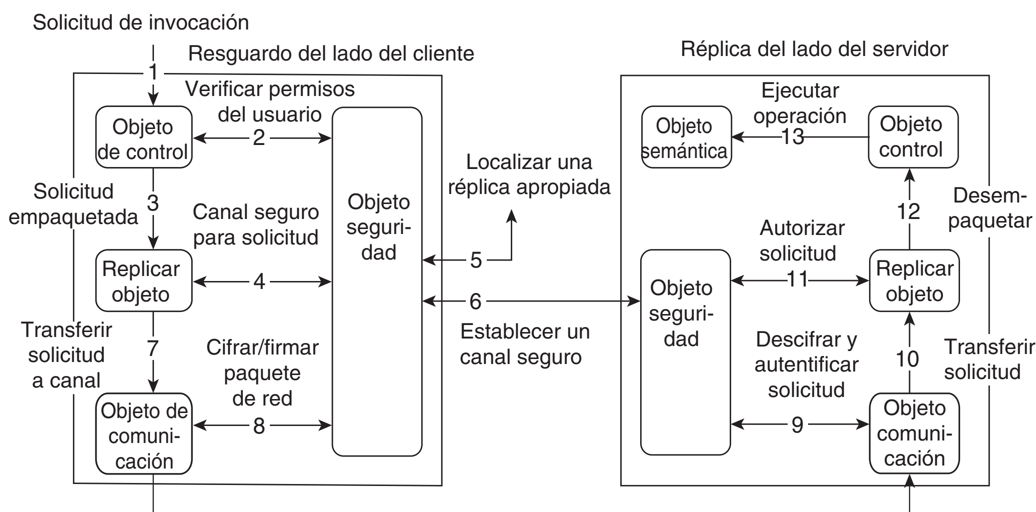


Figura 10-22. Invocación segura a un método en GLOBE.

mada local al método asociado, de igual modo que se llama a un procedimiento en una RPC.

2. El subobjeto de control verifica los permisos de usuario con la información guardada en el objeto de seguridad local. En este caso, el objeto de seguridad deberá tener un certificado de usuario válido.
3. La solicitud es empaquetada y transferida.
4. El subobjeto de replicación solicita al middleware que establezca un canal seguro para implementar una réplica apropiada.
5. El objeto de seguridad primero inicia la búsqueda de una réplica. Para alcanzar este objetivo, podría utilizar cualquier servicio de asignación de nombres capaz de buscar réplicas que haya sido certificado como capaz de ejecutar ciertos métodos. El servicio de localización Globe ha sido modificado para manejar tales búsquedas (Ballintijn, 2003).
6. Una vez que encuentra un réplica apropiada, el subobjeto de seguridad puede establecer un canal seguro con su contraparte, tras de lo cual el control es devuelto al subobjeto de replicación. Advierta que parte de este establecimiento requiere que la réplica compruebe que tiene permiso de realizar la invocación solicitada.
7. La solicitud se transfiere ahora al subobjeto de comunicación.
8. El subobjeto de comunicación cifra y firma la solicitud de modo que pueda transferirla a través de un canal.
9. Después de su recepción, la solicitud es descifrada y autentificada.
10. Entonces la solicitud es simplemente transferida al subobjeto de replicación del lado del servidor.
11. La autorización ocurre: en este caso, el certificado de usuario del resguardo del lado del cliente ha sido transferido a la réplica para que verifique si la solicitud puede realmente ser ejecutada.
12. Entonces la solicitud es desorganizada.
13. Al final, la operación puede ser ejecutada.

Aun cuando puede parecer que este procedimiento implica un gran número de pasos, el ejemplo muestra cómo una invocación a un método seguro puede ser dividida en pequeñas unidades, siendo cada unidad necesaria para garantizar que un cliente autentificado pueda realizar una invocación autorizada en una réplica autentificada. Virtualmente, todos los sistemas distribuidos basados en objetos siguen estos pasos. La diferencia con Globe es que se debe localizar una réplica apropiada, y que ésta tiene que comprobar que puede ejecutar la invocación al método. Se deja tal comprobación como ejercicio para el lector.

10.8.2 Seguridad para objetos remotos

Cuando se utilizan objetos remotos, a menudo se advierte que la referencia a ellos se implementa como un resguardo completo del lado del cliente, la cual contiene toda la información requerida para acceder al objeto remoto. En su forma más simple, la referencia contiene la dirección de contacto exacta del objeto y utiliza un empaquetamiento estándar y protocolos de comunicación para enviar la invocación al objeto remoto.

Sin embargo, en sistemas como Java, el resguardo del lado del cliente (llamada **proxy**) puede ser virtualmente cualquier cosa. La idea básica es que el desarrollador de un objeto remoto desarrolle también el proxy y que, posteriormente, lo registre con un servicio de directorio. Cuando un cliente esté buscando un objeto, eventualmente se pondrá en contacto con el servicio de directorio, recuperará el proxy, y lo instalará. Desde luego, existen algunos problemas serios relacionados con este método.

En primer lugar, cuando el servicio de directorio es secuestrado, un atacante puede ser capaz de regresar un proxy falso al cliente. Dicho proxy puede realmente poner en riesgo toda la comunicación entre el cliente y el servidor que aloja el objeto remoto, dañándolos a ambos.

En segundo lugar, el cliente no tiene forma de autenticar el servidor: sólo tiene el proxy, y toda la comunicación con el servidor pasa necesariamente a través de éste. Esta situación puede ser indeseable, sobre todo porque el cliente ahora simplemente necesita confiar en que el proxy realizará su trabajo correctamente.

Asimismo, puede ser más difícil que el servidor autentifique al cliente. La autentificación puede ser necesaria cuando se envía información sensible al cliente. Además, como la autentificación del cliente ahora está vinculada al proxy, también se puede tener la situación de que un atacante esté haciendo una simulación de un cliente y dañe al objeto remoto.

Li y colaboradores (2004b) describen una arquitectura de seguridad general que puede ser utilizada para hacer más seguras las invocaciones a objetos remotos. En su modelo, asumen que los proxies en realidad son provistos por el desarrollador de un objeto remoto y registrados con un servicio de directorio. Este método se sigue en la RMI de Java, pero también en Jini (Sun Microsystems, 2005).

El primer problema a resolver es autentificar un objeto remoto. En su solución, Li y Mitchell proponen un método de dos pasos. Primero, el proxy descargado de un servicio de directorio es firmado por el objeto remoto y permite que el cliente verifique su origen. El proxy, a su vez, autentificará al objeto utilizando TLS con autentificación de servidor, como se vio en el capítulo 9. Observemos que es tarea del desarrollador del objeto asegurarse de que el proxy autentifique apropiadamente al objeto. El cliente deberá confiar en este comportamiento, puesto que es capaz de autentificar el proxy, confiando en que la autentificación del objeto está al mismo nivel que confiar en que el objeto remoto se comporte decentemente.

Para autentificar al cliente, se utiliza una autentificación aparte. Cuando un cliente esté buscando el objeto remoto será dirigido a este autentificador, a partir del cual descargará un **proxy de autentificación**. Éste es un proxy especial que ofrece una interfaz mediante la cual el cliente puede hacer que sea autentificado por el objeto remoto. Si esta autentificación tiene éxito, entonces el objeto remoto (o en realidad, servidor de objetos) transferirá el proxy al cliente. Observe que este método permite la autentificación independiente del protocolo utilizado por el proxy, lo cual se considera una importante ventaja.

Otra ventaja destacable de separar la autentificación del cliente es que ahora es posible transferir proxies dedicados a clientes. Por ejemplo, a ciertos clientes se les puede permitir que ejecuten únicamente métodos de sólo lectura. En ese caso, después de que ocurra la autentificación, el cliente será transferido a un proxy que ofrece sólo tales métodos, y ningunos otros. Un control de acceso más refinado puede ser fácilmente contemplado.

10.9 RESUMEN

La mayoría de los sistemas distribuidos basados en objetos utilizan un modelo de objetos remotos en el cual un objeto está alojado en un servidor que permite a clientes remotos invocar métodos. En muchos casos, estos objetos se construirán en tiempo de ejecución, lo cual efectivamente significa que su estado y también posiblemente su código se carguen en un servidor cuando un cliente realice una invocación remota. Globe es un sistema en el cual objetos compartidos verdaderamente distribuidos son soportados. En este caso, el estado de un objeto puede estar físicamente distribuido y replicado a través de múltiples máquinas.

Para soportar objetos distribuidos, es importante separar la funcionalidad de propiedades extrafuncionales como la tolerancia a fallas y la escalabilidad. Con este propósito, se han desarrollado servidores avanzados de objetos para alojar objetos. Un servidor de objetos proporciona muchos servicios a objetos básicos, incluidos medios para guardar objetos o para garantizar la serialización de solicitudes entrantes. Otro rol importante es proporcionar al mundo externo la ilusión de que un conjunto de datos y procedimientos que operan con dichos datos corresponden al concepto de un objeto. Este rol se implementa por medio de adaptadores de objetos.

Cuando se trata de comunicación, la forma prevaleciente de invocar un objeto es por medio de una invocación a un método remoto (RMI), la cual es muy similar a una RPC. Una importante diferencia es que los objetos distribuidos proporcionan, en general, una referencia a objetos a nivel de todo el sistema, lo cual permite a un proceso acceder a un objeto desde cualquier máquina. La referencia global a objetos resuelve muchos de los problemas de transferencia de parámetros que obstaculizan la transparencia de las RPC.

Existen muchas formas diferentes en las que estas referencias a objetos pueden ser implementadas, y van desde simples estructuras de datos pasivas que describen con precisión dónde puede ser contactado un objeto remoto, hasta códigos portátiles que simplemente necesitan ser invocados por un cliente. El segundo método ahora es adoptado comúnmente para implementar RMI Java.

En la mayoría de los sistemas no existen medidas especiales para manejar la sincronización de objetos. Una importante excepción es la forma en que son tratados los métodos sincronizados de Java: la sincronización ocurre sólo entre clientes que funcionan en la misma máquina. Los clientes que funcionan en máquinas diferentes tienen que tomar medidas especiales de sincronización. Estas medidas no forman parte del lenguaje Java.

La consistencia de entrada es un evidente modelo de consistencia para objetos distribuidos, y (a menudo) es implícitamente soportada en muchos sistemas. Es evidente ya que se puede asociar de forma natural un bloqueo distinto para cada objeto. Uno de los problemas que resultan de la

replicación de objetos son las invocaciones replicadas. Este problema es más evidente porque los objetos tienden a ser tratados como cajas negras.

La tolerancia a fallas en sistemas basados en objetos distribuidos sigue muy de cerca los métodos utilizados en otros sistemas distribuidos. Una excepción se forma cuando se trata de hacer que la máquina virtual de Java sea tolerante a fallas al permitir que opere como máquina determinista de estado finito. Entonces, replicando varias de estas máquinas, se obtiene una forma natural de proporcionar tolerancia a fallas.

La seguridad para objetos distribuidos evoluciona en torno a la idea de soportar invocaciones a métodos seguros. Un ejemplo amplio que generaliza estas invocaciones a objetos replicados es Globe. Como resulta ser, es posible separar limpiamente las políticas de los mecanismos. Esto es verdad tanto para autentificación como para autorización. Debemos prestar una atención especial a sistemas en los cuales se requiere que el cliente descargue un proxy de un servicio de directorio, como comúnmente es el caso para Java.

PROBLEMAS

1. Se hizo una distinción entre objetos remotos y objetos distribuidos. ¿Cuál es la diferencia?

2. ¿Por qué es útil definir las interfaces de un objeto en un lenguaje de definición de interfaz?

3. Algunas implementaciones de sistemas middleware de objetos distribuidos están enteramente basadas en invocaciones dinámicas a métodos. Incluso las invocaciones estáticas se compilan para transformarlas en dinámicas. ¿Cuál es el beneficio de este método?

4. Describa un protocolo simple que implemente semántica de, cuando mucho, una vez para la invocación a un método.

5. ¿Deberán ser persistentes los objetos del lado de cliente y del lado del servidor para invocaciones a métodos asíncronos?

6. En el texto se mencionó que una implementación de la invocación a un método asíncrono de CORBA no afecta la implementación del lado del servidor de un objeto. Explique por qué éste es el caso.

7. Proporcione un ejemplo en el cual el uso (inadvertido) de mecanismos de llamada automática puede conducir con facilidad a una situación indeseable.

8. ¿Es posible que un objeto tenga más de un sirviente?

9. ¿Es posible tener implementaciones específicas de sistema de referencias a objetos CORBA mientras aún se es capaz de intercambiar referencias con otros sistemas basados en CORBA?

10. ¿Cómo se pueden autentificar las direcciones de contacto regresadas por un servicio de búsqueda de objetos Globe seguros?

11. ¿Cuál es la diferencia clave entre referencias a objetos en CORBA y en Globe?

12. Considere Globe. Describa un protocolo simple mediante el cual se establece un canal seguro entre un proxy de usuario (el cual tiene acceso a la clave privada de Alicia) y una réplica que se sabe con seguridad puede ejecutar un método dado.

13. Proporcione un ejemplo de una implementación de una referencia a un objeto que permita a un cliente vincularse a un objeto remoto transitorio.

14. Java y otros lenguajes soportan excepciones, las cuales surgen cuando ocurre un error. ¿Cómo implementaría usted excepciones en RPC y RMI?

15. ¿Cómo incorporaría usted una comunicación asíncrona persistente al modelo de comunicación basado en RMI a objetos remotos?

16. Considere un sistema distribuido basado en objetos que soporta replicación de objetos, en el cual *todas* las invocaciones a métodos se realizan en completo orden. También suponga que la invocación a un método es atómica (por ejemplo, porque cada objeto se bloquee automáticamente cuando es invocado). ¿Semejante sistema proporciona consistencia de entradas? ¿Qué hay en cuanto a consistencia secuencial?

17. Describa un esquema basado en el destinatario para ocuparse de invocaciones replicadas, como se menciona en el texto.

11

SISTEMAS DE ARCHIVO DISTRIBUIDOS

Si consideramos que el compartir datos es fundamental en los sistemas distribuidos, no sorprende que los sistemas de archivo compartidos constituyan el fundamento de muchas aplicaciones. Los sistemas de archivo distribuidos permiten que múltiples procesos compartan datos durante largos periodos en forma segura y confiable. Así, han sido utilizados como la capa básica de aplicaciones y sistemas distribuidos. En este capítulo, los sistemas de archivo distribuidos se consideran como un paradigma para sistemas distribuidos de propósito general.

11.1 ARQUITECTURA

Nuestro análisis de los sistemas de archivo distribuidos se inicia viendo cómo están organizados en general. La mayoría se construye siguiendo una arquitectura cliente-servidor tradicional, aunque también existen soluciones descentralizadas. A continuación, examinaremos ambas clases de organizaciones.

11.1.1 Arquitecturas cliente-servidor

Muchos sistemas de archivo distribuidos se organizan a lo largo de las líneas de arquitecturas cliente-servidor, siendo el **sistema de archivo de red** (**NFS**, por sus siglas en inglés) de Sun Microsystem una de las más ampliamente utilizadas en los sistemas basados en UNIX. Se tomará el NFS como un ejemplo canónico de sistemas de archivo distribuidos basados en servidor a lo largo de todo este

capítulo. En particular nos concentraremos en el NFSv3, la tercera versión ampliamente utilizada del NFS (Callaghan, 2000) y el NFSv4, la cuarta versión más reciente (Shepler y cols., 2003). También estudiaremos sus diferencias.

La idea básica detrás del NFS es que cada servidor de archivos proporcione una visión estandarizada de su sistema de archivo local. En otros términos, no importa cómo se implemente el sistema de archivo local, cada servidor NFS soporta el mismo modelo. Este método también ha sido adoptado por otros sistemas de archivo. El NFS cuenta con un protocolo de comunicación que permite a los clientes acceder a archivos guardados en un servidor; por tanto, es posible que un conjunto heterogéneo de procesos, quizá ejecutándose con sistemas operativos y máquinas diferentes, compartan un sistema de archivo común.

El modelo que sirve de fundamento al NFS y a sistemas similares es de un **servicio de archivos remoto**. Este modelo ofrece a los clientes un acceso transparente a un sistema de archivo gestionado por un servidor remoto. Sin embargo, los clientes normalmente desconocen la ubicación de los archivos. En cambio, disponen de una interfaz para que interactúen con el sistema de archivo similar a la interfaz ofrecida por un sistema de archivo convencional. En particular, al cliente sólo se le ofrece una interfaz que contiene varias operaciones de archivo, pero el servidor es responsable de implementarlas. Por consiguiente, a este modelo también se le denomina **modelo de acceso remoto**. Se muestra en la figura 11-1(a).

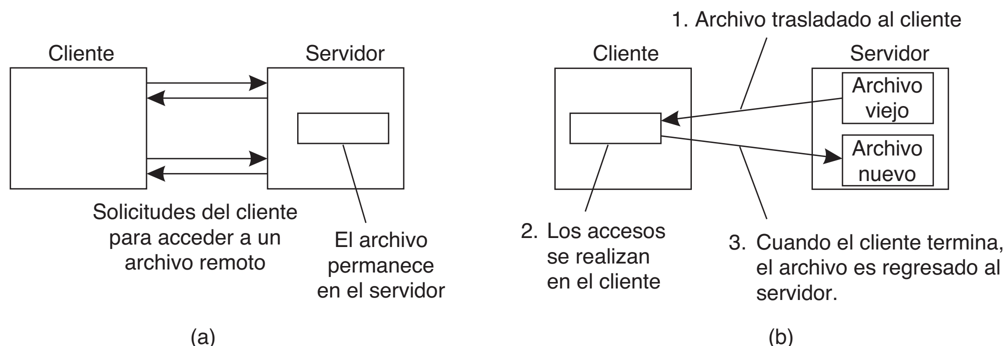


Figura 11-1. (a) Modelo de acceso remoto. (b) Modelo de carga y descarga.

Por contraste, en el **modelo de carga y descarga** un cliente accede a un archivo localmente después de haberlo descargado del servidor, como se muestra en la figura 11-1(b). Cuando el cliente termina con el archivo, lo carga otra vez en el servidor para que pueda ser utilizado por otro cliente. El servicio FTP de internet se utiliza de esta manera cuando un cliente descarga un archivo completo, lo modifica y luego lo repone.

El NFS ha sido implementado para un gran número de sistemas operativos, aunque predominan las versiones basadas en UNIX. Para virtualmente todos los sistemas UNIX modernos, el NFS generalmente se implementa siguiendo la arquitectura en capas mostrada en la figura 11-2.

Un cliente accede al sistema de archivo usando las invocaciones a sistema provistas por su sistema operativo local. Sin embargo, la interfaz del sistema de archivo UNIX es reemplazada por

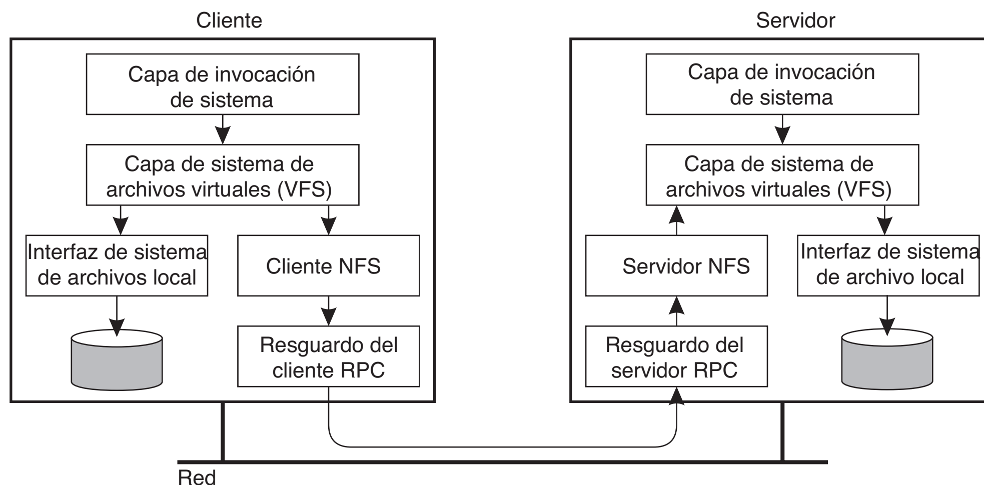


Figura 11-2. Arquitectura NFS básica para sistemas UNIX.

una interfaz para comunicarse con el **sistema de archivo virtual** (**VFS**, por sus siglas en inglés), el cual por ahora es efectivamente el estándar para comunicarse con sistemas de archivo (distribuidos) diferentes (Kleiman, 1986). Virtualmente todos los sistemas operativos modernos proporcionan un VFS, y de no ser así los desarrolladores se ven obligados a reimplementar un sistema operativo enorme cuando adoptan una nueva estructura de sistema de archivo. Con el NFS, las operaciones de interfaz se transfieren o a un sistema de archivo local o a otro componente conocido como **cliente NFS**, el cual se encarga de manejar el acceso a los archivos guardados en un servidor remoto. En el NFS, toda la comunicación entre cliente y servidor se realiza mediante RPC. El cliente NFS implementa las operaciones en el sistema de archivo NFS como RPC al servidor. Observe que las operaciones ofrecidas por la interfaz VFS pueden ser diferentes de aquellas ofrecidas por el cliente NFS. La idea completa del VFS es ocultar las diferencias entre varios sistemas de archivo.

Del lado del servidor, la organización es similar. El **servidor NFS** se encarga de manejar las solicitudes de clientes. El resguardo de RPC desempaqueta las solicitudes y el servidor NFS las transforma en operaciones con archivos VFS regulares que posteriormente se transfieren a la capa VFS. De nueva cuenta, el VFS es responsable de implementar un sistema de archivo local donde los archivos están guardados.

Una importante ventaja de este esquema es que el NFS es independiente en gran medida de los sistemas de archivo locales. En principio, realmente no importa si el sistema operativo en el lado del cliente o del servidor implementa un sistema de archivo UNIX, un sistema de archivo Windows 2000, o incluso un viejo sistema de archivo MS-DOS. Lo único importante es que estos sistemas de archivo cumplan con el modelo de sistema de archivo ofrecido por el NFS. Por ejemplo, el MS-DOS con sus cortos nombres de archivo no puede ser utilizado para implementar un servidor NFS de una manera totalmente transparente.

Modelo de sistema de archivo

El modelo de sistema de archivo ofrecido por el NFS casi es el mismo que el ofrecido por sistemas basados en UNIX. Los archivos se tratan como secuencias de bytes no interpretadas. Están jerárquicamente organizados en un grafo de nombres donde los nodos representan directorios y archivos. El NFS también soporta vínculos duros, así como también vínculos simbólicos, igual que cualquier sistema de archivo UNIX. A los archivos se les asigna un nombre, pero se accede a ellos por medio de un **manejador de archivo** como el de UNIX, el cual analizamos detalladamente a continuación. En otros términos, para acceder a un cliente, primero se busca su nombre en un servicio de asignación de nombres y se obtiene el manejador de archivo asociado. Además, cada archivo tiene cierto número de atributos cuyos valores pueden ser buscados y cambiados. Más adelante en este capítulo regresamos a la asignación de nombres de archivo.

La figura 11-3 muestra las operaciones generales con archivos soportadas por las versiones 3 y 4 de NFS, respectivamente. La operación create se utiliza para crear un archivo, aunque sus significados son un tanto diferentes en NFSv3 y NFSv4. En la versión 3, la operación sirve para crear archivos regulares. Con operaciones distintas se crean archivos especiales. La operación link es para crear vínculos duros; con Symlink se crean vínculos simbólicos. Mkdir crea subdirectorios. Archivos especiales, como archivos de dispositivos, sockets y pipes (tuberías) nombrados se crean por medio de la operación mknod.

Esta situación cambia por completo en NFSv4, donde create se utiliza para crear archivos *no regulares*, los cuales incluyen vínculos simbólicos, directorios y archivos especiales. Los vínculos duros se siguen creando con una operación link distinta, pero los archivos regulares se crean con la operación open, la cual es nueva en NFS y es una desviación importante del método de manejo de archivos en versiones más viejas. Hasta la versión 4, NFS estaba diseñado para permitir que sus servidores de archivos fueran sin estado. Por razones que se verán más adelante en este capítulo, este criterio de diseño ha sido abandonado en NFSv4, en donde se supone que los servidores generalmente mantendrán su estado entre operaciones con el mismo archivo.

La operación rename se utiliza para cambiar el nombre de un archivo existente como en UNIX.

Los archivos se eliminan por medio de la operación remove. En la versión 4, esta operación se utiliza para eliminar cualquier clase de archivo. En versiones previas, se necesitaba una operación rmdir para eliminar un subdirectorio. Un archivo se eliminaba por su nombre y el número de vínculos duros se reducía en uno. Si el número de vínculos se reduce a cero, el archivo puede ser destruido.

La versión 4 permite que los clientes abran y cierren archivos (regulares). La apertura de un archivo no existente tiene el efecto colateral de que se crea un archivo nuevo. Para abrir un archivo, un cliente proporciona el nombre, junto con varios valores para atributos. Por ejemplo, un cliente puede especificar qué archivo deberá ser abierto para acceso de escritura. Después de que un archivo se ha abierto con éxito, un cliente puede acceder a él por medio de su manejador. Éste se utiliza también para cerrar el archivo, mediante lo cual el cliente le dice al servidor que ya no necesitará tener acceso al archivo. El servidor, a su vez, puede liberar cualquier estado en el que se encuentre para que el cliente acceda al archivo.

Operación	**v3**	**v4**	**Descripción**
Create	Sí	No	Crea un archivo regular
Create	No	Sí	Crea un archivo no regular
Link	Sí	Sí	Crea un vínculo duro a un archivo
Symlink	Sí	No	Crea un vínculo simbólico a un archivo
Mkdir	Sí	No	Crea un subdirectorio en un directorio dado
Mknod	Sí	No	Crea un archivo especial
Rename	Sí	Sí	Cambia el nombre de un archivo
Remove	Sí	Sí	Elimina un archivo de un sistema de archivos
Rmdir	Sí	No	Elimina un subdirectorio vacío de un directorio
Open	No	Sí	Abre un archivo
Close	No	Sí	Cierra un archivo
Lookup	Sí	Sí	Busca un archivo por su nombre
Readdir	Sí	Sí	Lee las entradas en un directorio
Readlink	Sí	Sí	Lee el nombre de ruta guardado en un vínculo simbólico
Getattr	Sí	Sí	Establece valores de atributo para un archivo
Setattr	Sí	Sí	Establece uno o más valores de atributo para un archivo
Read	Sí	Sí	Lee los datos contenidos en un archivo
Write	Sí	Sí	Escribe datos en un archivo

Figura 11-3. Lista incompleta de operaciones incluidas en un sistema de archivo soportadas mediante NFS.

La operación lookup se utiliza para buscar un majejador de archivo para un nombre de ruta dado. En NFSv3, la operación de búsqueda no resolverá un nombre más allá de un punto de montaje. (Recordemos del capítulo 5 que el punto de montaje es un directorio que en esencia representa un vínculo a un subdirectorio en un espacio de nombre *foreign*). Por ejemplo, supongamos que el nombre */remote/vu* se refiere a un punto de montaje localizado en una gráfica de asignación de nombres. Cuando se resuelve el nombre */remote/vu/mbox*, la operación de búsqueda en NFSv3 regresará el manejador de archivo correspondiente al punto de montaje */remote/vu* junto con el resto del nombre de la ruta (es decir, *mbox*). Luego se requiere que el cliente monte explícitamente el sistema de archivo requerido para completar la búsqueda del nombre. En este contexto, un sistema de archivo es el conjunto de archivos, atributos, directorios y bloques de datos que se implementan juntos como un dispositivo de bloques lógico (Tanenbaum y Woodhull, 2006).

En la versión 4, las cosas se simplificaron. En este caso, la operación lookup intentará resolver el nombre completo, incluso si esto implica cruzar los puntos de montaje. Observe que aplicar este método sólo es posible si ya se montó el sistema de archivo en los puntos de montaje. El cliente es capaz de detectar que dicho punto de montaje ha sido cruzado al inspeccionar el identificador del sistema de archivo que posteriormente es regresado cuando se completa la búsqueda.

Existe una operación readdir por separado para leer las entradas en un directorio. Esta operación regresa una lista de pares (*nombre*, *manejador de archivo*) junto con valores de atributo que el

cliente solicitó. El cliente puede especificar también cuántas entradas deberá devolver. La operación regresa un desplazamiento (offset) que puede ser utilizado en una invocación subsiguiente a readdir para leer la siguiente serie de entradas.

La operación readlink se utiliza para leer los datos asociados con un vínculo simbólico. Normalmente, estos datos corresponden a un nombre de ruta que posteriormente puede ser buscada. Observe que la operación lookup no puede manejar vínculos simbólicos. En cambio, cuando se llega a un vínculo simbólico, la resolución del nombre se detiene y el cliente tiene que invocar primero readlink para indagar dónde deberá continuar la resolución del nombre.

Los archivos tienen varios atributos asociados. De nueva cuenta, existen diferencias importantes entre las versiones 3 y 4 de NFS, las cuales analizaremos con mayor detalle más adelante. Atributos típicos incluyen el tipo de archivo (para indicar si se trata de un directorio, un vínculo simbólico, un archivo especial, etc.), la longitud del archivo, el identificador del sistema que contiene el archivo y la última vez que éste fue modificado. Mediante las operaciones getattr y setattr se leen y establecen, respectivamente, los atributos de archivo.

Por último, existen operaciones para leer y escribir datos en un archivo. La lectura de datos por medio de la operación read es muy simple. El cliente especifica el desplazamiento y el número de bytes que va a leer. El cliente recibe el número real de bytes leídos, junto con información adicional sobre el estado (por ejemplo, si se llegó al final del archivo).

La escritura de datos en un archivo se realiza con la operación write. El cliente de nuevo especifica en el archivo la posición donde se va a iniciar la escritura, la cantidad de bytes que se va a escribir y los datos. Además, se puede instruir al servidor para que garantice que todos los datos se escriban en un almacenamiento estable (en el capítulo 8 presentamos el almacenamiento estable). Se requiere que los servidores NFS soporten dispositivos de almacenamiento que puedan sobrevivir a fallas de corriente, fallas del sistema operativo y fallas de hardware.

11.1.2 Sistemas de archivo distribuido basados en cluster

El NFS es un ejemplo típico de muchos sistemas de archivo distribuidos, los cuales generalmente se organizan con arreglo a una arquitectura cliente-servidor tradicional. Esta arquitectura a menudo se mejora agrupando los servidores con pocas diferencias.

Si consideramos que los grupos de servidores a menudo se utilizan en aplicaciones en paralelo, no sorprende que sus sistemas de archivo asociados se ajusten como corresponde. Una técnica muy conocida es desplegar **técnicas de distribución de archivos**, mediante las cuales un archivo se distribuye a través de múltiples servidores. La idea básica es simple: distribuyendo un archivo grande entre múltiples servidores, es posible buscar sus diferentes partes en paralelo. Desde luego, semejante organización funciona bien sólo si la aplicación se organiza de tal forma que el acceso a los datos en paralelo tenga sentido. Esto requiere generalmente que los datos, tal como se guardan en el archivo, tengan una estructura muy regular, por ejemplo, una matriz (densa).

Para aplicaciones de propósito general, o para aquellas con tipos irregulares o muchos tipos de estructuras de datos, la distribución de archivos puede no ser una herramienta efectiva. En esos casos, a menudo es más conveniente particionar el sistema de archivo en su totalidad y simplemente

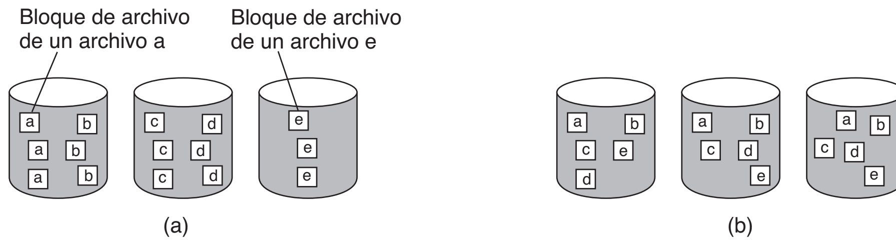


Figura 11-4. Diferencia entre (a) distribución de archivos completos a través de varios servidores y (b) fragmentación de archivos para acceso paralelo.

guardar los diferentes archivos en diferentes servidores, pero no particionar un solo archivo a través de múltiples servidores. La diferencia entre estos dos métodos se muestra en la figura 11-4.

Más interesantes son los casos en que se debe organizar un sistema de archivos distribuidos para grandes centros de datos, tales como los utilizados por compañías como Amazon y Google. Estas compañías ofrecen servicios a clientes web cuyo resultado son lecturas y actualizaciones de un número masivo de archivos distribuidos a través de literalmente decenas de miles de computadoras [vea también Barroso y cols. (2003)]. En tales entornos, la suposiciones tradicionales en relación con sistemas de archivo distribuidos ya no son válidas. Por ejemplo, es de esperarse que en cualquier momento falle una computadora.

Para abordar estos problemas, Google, por ejemplo, desarrolló su propio **sistema de archivo Google** (**GFS**, por sus siglas en inglés), cuyo diseño se describe en Ghemawat y colaboradores (2003). Los archivos Google tienden a ser muy grandes, comúnmente de varios gigabytes, y cada uno contiene muchos objetos más pequeños. Además, las actualizaciones de los archivos generalmente se realizan anexando datos en lugar de sobreescribir algunas partes de un archivo. Estas observaciones, junto con el hecho de que las fallas de servidor son la norma en lugar de la excepción, llevaron a la construcción de grupos de servidores tal como se muestra en la figura 11-5.

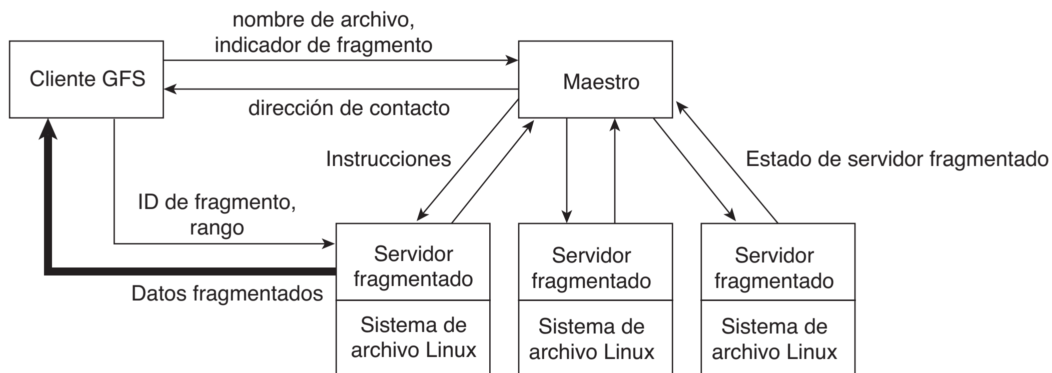


Figura 11-5. Organización de un grupo de servidores Google.

Cada grupo de GFS se compone de un servidor maestro junto con múltiples **servidores fragmentados**. Cada archivo GFS es dividido en fragmentos de 64 Mbytes cada uno, tras de lo cual es-

tos fragmentos se distribuyen a través de los servidores llamados fragmentados. Una observación importante es que un GFS maestro se contacta sólo para recabar información de metadatos. En particular, un cliente GFS transfiere un nombre de archivo y un índice de fragmento al maestro, y espera una dirección de contacto para el fragmento. La dirección de contacto contiene toda la información necesaria para acceder al servidor fragmentado correcto y obtener el fragmento de archivo requerido.

Con esta finalidad, el GFS maestro esencialmente mantiene un espacio de nombre, adjunto a un mapeo desde un nombre de archivo hasta los fragmentos. Cada fragmento lleva un identificador asociado para que el servidor fragmentado pueda buscarla. Además, el servidor maestro no pierde de vista la ubicación de un fragmento. Los fragmentos se replican en prevención de fallas de manejador, pero nada más para eso. Una característica interesante es que el GFS maestro no intenta llevar la cuenta precisa de las ubicaciones de los fragmentos. En cambio, de vez en cuando se pone en contacto con los servidores fragmentados para ver qué fragmentos tienen guardados.

La ventaja de este esquema es la simplicidad. Observemos que el servidor maestro controla la asignación de los fragmentos a los servidores fragmentados. Además, éstos llevan la cuenta de lo que tienen guardado. Por consiguiente, una vez que el servidor maestro obtiene las ubicaciones de los fragmentos, tiene una imagen precisa de dónde están guardados los datos. Sin embargo, las cosas se complicarían si esta visión tuviera que ser consistente todo el tiempo. Por ejemplo, cada vez que un servidor fragmentado se congelara o cuando se agregara un servidor, el servidor maestro tendría que ser informado. En cambio, es mucho más simple refrescar su información a partir del conjunto actual de servidores fragmentados mediante encuestas. Los clientes GFS simplemente necesitan saber qué servidores fragmentados cree el servidor maestro que están guardando los datos solicitados. Como los fragmentos se replican de todos modos, existe una alta probabilidad de que un fragmento esté disponible al menos en uno de los servidores fragmentados.

¿Por qué se agranda este esquema? Un importante tema de diseño es que el servidor maestro está en control en gran medida, pero que no forma un cuello de botella a causa del trabajo que necesita realizar. Se han tomado dos tipos de medidas para manejar la escalabilidad.

La primera medida, y con mucho la más importante, es que la mayor parte del trabajo sea realizada por servidores fragmentados. Cuando un cliente necesita acceder a datos, se pone en contacto con el servidor maestro para indagar qué servidores contienen los datos. Después de eso, se comunica sólo con los servidores fragmentados. Los fragmentos se replican de acuerdo con un esquema de respaldo primario. Cuando el cliente está realizando una operación de actualización, se pone en contacto con el servidor fragmentado más cercano que contiene los datos y dirige sus actualizaciones a dicho servidor. Este servidor envía la actualización al siguiente más cercano que contiene los datos y así sucesivamente. Una vez que todas las actualizaciones se han propagado, el cliente se pondrá en contacto con el servidor fragmentado primario, el cual asignará entonces un número secuencial a la operación de actualización y la transferirá a los respaldos. Entre tanto, el servidor maestro se mantiene fuera del bucle.

La segunda medida, el nombre de espacio (jerárquico) para archivos, se implementa con una tabla de nivel único simple, en la cual los nombres de ruta se dirigen a metadatos (equivalentes a los nodos de los sistemas de archivo tradicionales). Además, toda la tabla se mantiene en la memoria principal, junto con la dirección de los archivos ubicados en las secciones. Las actualizaciones de estos datos se guardan en un almacenamiento persistente. Cuando el registro se vuelve demasiado

grande, se utiliza un punto de control mediante el cual los datos que se encuentran en la memoria principal se guardan de tal forma que puedan ser tipografiados de vuelta en la memoria principal. Por consiguiente, la intensidad de E/S de un GFS maestro se reduce fuertemente.

Esta organización permite que un solo servidor maestro controle algunos cientos de servidores fragmentados, lo cual es una cantidad considerable para un solo grupo. Con la organización subsiguiente de un servicio tal como Google en servicios más pequeños tipografiados en grupos, no es difícil imaginar que se puede hacer que un enorme número de grupos funcionen juntos.

11.1.3 Arquitecturas simétricas

Desde luego, también existen organizaciones totalmente simétricas basadas en técnicas punto a punto. Todas las propuestas actuales utilizan un sistema basado en DHT de distribución de datos, combinado con un mecanismo de búsqueda basado en una clave. Una importante diferencia es decidir si construyen un sistema de archivos sobre una capa de almacenamiento distribuido, o si archivos completos deben guardarse en los nodos participantes.

Un ejemplo del primer tipo de sistema de archivo es Ivy, un sistema de archivo distribuido construido con un sistema basado en cuerdas DHT. Ivy se describe en Muthitacharoen y colaboradores (2002). En esencia, su sistema se compone de tres capas tal como se muestra en la figura 11-6. La capa más baja está formada por un sistema de cuerdas que proporciona operaciones de búsqueda descentralizadas básicas. La capa de en medio es una capa de almacenamiento totalmente orientada a bloques distribuidos. Finalmente, en la parte de arriba se encuentra una capa que implementa un sistema de archivo a la manera del NFS.

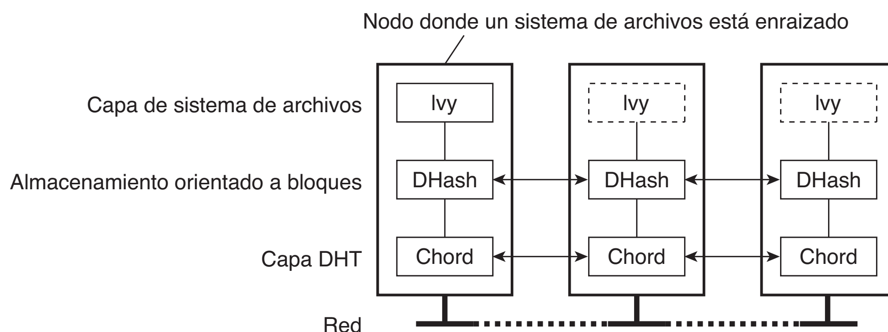


Figura 11-6. Organización del sistema de archivo distribuido Ivy.

En Ivy el almacenamiento de datos es realizado por un sistema de almacenamiento distribuido orientado a bloques llamado **DHash** (Dabek y cols., 2001). En esencia, el sistema DHash es bastante simple. Sólo maneja bloques de datos, cada bloque es de 8 KB. Ivy utiliza dos clases de bloques de datos. Un **bloque que contiene un hash** tiene una clave asociada, el cual se calcula como el hash seguro del contenido del bloque. De esta manera, siempre que se busca un bloque, un cliente puede verificar de inmediato si ha buscado el bloque correcto, o si le fue regresada otra versión o una versión corrupta. Además, Ivy también utiliza **bloques de clave pública**, los cuales son bloques que

tienen una clave pública como clave de búsqueda y cuyo contenido ha sido firmado con la clave privada asociada.

Para incrementar la disponibilidad, DHash replica cada bloque *B* a los *k* sucesores inmediatos del servidor responsable de guardar *B*. Además, los bloques buscados también se guardan en la memoria caché junto con la ruta que siguió la solicitud de búsqueda.

Los archivos se implementan como una estructura de datos aparte encima del sistema DHash. Para alcanzar este objetivo, cada usuario mantiene un registro de las operaciones que realiza en los archivos. Por simplicidad, suponemos que existe sólo un usuario por nodo de modo que cada nodo tenga su propio registro. Un registro es una lista vinculada de registros inmutables, donde cada registro contiene toda la información relacionada con una operación del sistema de archivo Ivy. Cada nodo anexa registros sólo a su propio registro local. Sólo un encabezado de los registros es mutable y apunta a los registros agregados más recientemente. Cada registro se guarda en un bloque de contenido de hash distinto, en tanto que el encabezado del registro se guarda en un bloque de clave pública.

Existen diferentes tipos de registros, que corresponden aproximadamente a las diferentes operaciones soportadas por NFS. Por ejemplo, cuando se actualiza una operación en un archivo, se crea un registro de *escritura* que contiene el identificador del archivo junto con el desplazamiento del apuntador de pila y los datos que se están escribiendo. Asimismo, existen registros para crear archivos (es decir, agregar un nuevo nodo índice), manipular directorios, etcétera.

Para crear un nuevo sistema de archivo, un nodo simplemente crea un nuevo registro junto con un nuevo nodo índice que servirá como raíz. Ivy despliega lo que se conoce como **servidor de bucle de retroceso NFS**, el cual solamente es un servidor a nivel de usuario local que acepta solicitudes NFS de clientes locales. En el caso de Ivy, este servidor NFS soporta el montaje de un sistema de archivos recién creados que permite a las aplicaciones acceder al sistema como cualquier otro sistema de archivo NFS.

Cuando se realiza una operación read, el servidor NFS Ivy local repasa el registro, recopilando los datos de los registros que representan operaciones write con el mismo bloque de datos, lo cual permite recuperar los valores más recientemente guardados. Observemos que como cada registro se guarda como un bloque DHash, puede ser que se requieran varias búsquedas a través de la red superpuesta para recuperar los valores relevantes.

En lugar de utilizar una capa de almacenamiento orientada a bloques distinta, diseños alternativos proponen distribuir los archivos completos en lugar de bloques de datos. Los desarrolladores de Kosha (Butt y cols., 2004) proponen distribuir archivos a un nivel de directorio específico. En su método, cada nodo tiene un punto de montaje llamado */kosha*, el cual contiene los archivos que van a ser distribuidos mediante un sistema basado en DHT. Distribuir archivos al nivel de directorio 1 significa que todos los archivos incluidos en un subdirectorio */kosha/a* se guardarán en el mismo nodo. Asimismo, la distribución al nivel 2 implica que todos los archivos guardados en un subdirectorio */kosha/a/aa* se guardan en el mismo nodo. Tomando la distribución a nivel 1 como ejemplo, el nodo responsable de guardar archivos bajo */kosha/a* se encuentra calculando el hash de *a* y tomándolo como clave en una búsqueda.

La desventaja potencial de este método es que un nodo puede quedarse sin espacio de disco para guardar todos los archivos contenidos en el subdirectorio responsable de hacerlo. De nueva cuenta, una solución simple es colocar una rama de dicho subdirectorio en otro nodo y crear un vínculo simbólico hacia donde esté guardada entonces la rama.

11.2 PROCESOS

Cuando se trata de procesos, los sistemas de archivo distribuidos no tienen propiedades inusuales. En muchos casos, habrá diferentes tipos de procesos cooperadores: servidores de almacenamiento y gestores de archivos, justo como ya describimos para las diversas organizaciones.

El aspecto más interesante con respecto a procesos de sistemas de archivo es si deberán o no ser sin estado. El NFS es un buen ejemplo que ilustra los compromisos. Una de sus características distintivas de larga duración (comparada con otros sistemas de archivo distribuidos) fue el hecho de que los servidores eran sin estado. En otros términos, el protocolo NFS no requería que los servidores mantuvieran cualquier estado de cliente. Este método se siguió en las versiones 2 y 3, pero fue abandonado en la versión 4.

La ventaja principal del método sin estado es la simplicidad. Por ejemplo, cuando un servidor sin estado se congela, esencialmente no hay necesidad de entrar a una fase de recuperación para llevarlo a un estado previo. Sin embargo, como explicamos en el capítulo 8, se sigue teniendo en cuenta que al cliente no se le pueden dar garantías sobre si en realidad su solicitud ha sido o no realizada.

El método sin estado en el protocolo NFS no siempre se puede seguir a cabalidad en implementaciones prácticas. Por ejemplo, no es fácil que un servidor sin estado bloquee un archivo. En el caso de NFS, se utiliza un gestor de bloqueo aparte para manejar esta situación. Asimismo, ciertos protocolos de autenticación (o autentificación) requieren que el servidor mantenga el estado en sus clientes. No obstante, los servidores NFS generalmente podían diseñarse de tal forma que sólo era necesario mantener muy poca información sobre los clientes. En su mayor parte, el esquema funcionaba adecuadamente.

A partir de la versión 4, se abandonó el método sin estado, aunque el nuevo protocolo fue diseñado de tal forma que un servidor no tiene que mantener mucha información sobre sus clientes. Además de las que se acaban de mencionar, existen otras razones para elegir un método con estado. Una razón muy importante es que se espera que el NFS versión 4 también funcione a través de redes de área amplia. Esto requiere que los clientes sean capaces de hacer un uso efectivo de los cachés, y a su vez demanda un protocolo de consistencia de caché eficiente. Esos protocolos a menudo funcionan mejor en colaboración con un servidor que mantenga algo de información sobre los archivos utilizados por sus clientes. Por ejemplo, un servidor puede asociar un contrato con cada archivo que entregue a un cliente, para prometerle un acceso exclusivo de lectura y escritura hasta que expire o sea renovado el contrato. Más adelante en este capítulo regresamos a estos temas.

La diferencia más evidente con las versiones previas es el soporte de la operación open. Además, el NFS soporta procedimientos de retorno de llamadas mediante los cuales un servidor puede realizar una RPC a un cliente. Claramente, los retornos de llamadas también requieren que un servidor no pierda de vista a sus clientes.

Un razonamiento similar ha afectado el diseño de otros sistemas de archivo distribuidos. Con mucho, resulta que mantener un diseño totalmente sin estado puede ser bastante difícil, y a menudo conduce a construir soluciones con estado como mejora, ese es el caso con el bloqueo de archivos NFS.

11.3 COMUNICACIÓN

Igual que con los procesos, no existe nada particularmente especial o inusual acerca de la comunicación en sistemas de archivo distribuidos. Muchos de estos sistemas están basados en llamadas a procedimientos remotos (RPC), aunque se hicieron algunas mejoras interesantes para soportar casos especiales. La razón principal para seleccionar un mecanismo de RPC es hacer que el sistema sea independiente de los sistemas operativos subyacentes, de redes, y de protocolos de transporte.

11.3.1 RPC en NFS

Por ejemplo, en NFS toda la comunicación entre un cliente y el servidor ocurre a lo largo de un protocolo **RPC de computación de red abierta** (**ONC RPC**, por sus siglas en inglés), el cual se define formalmente en Srinivasan (1995a), junto con un estándar para la representación de datos (Srinivasan, 1995b). El ONC RPC es similar a otros sistemas RPC tal como lo vimos en el capítulo 4.

Cada operación NFS puede ser implementada como una llamada a un procedimiento remoto única realizada a un servidor de archivos. En realidad, hasta el NFSv4, el cliente era responsable de hacer la vida del servidor tan fácil como fuera posible manteniendo las solicitudes relativamente simples. Por ejemplo, para leer datos de un archivo por primera vez, un cliente normalmente debía buscar el manejador del archivo con una operación lookup, después de lo cual podía emitir una solicitud read, como se muestra en la figura 11-7(a).

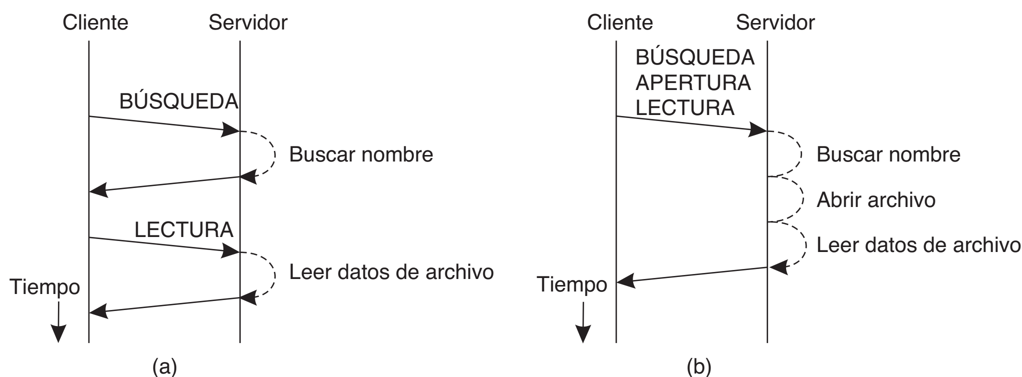


Figura 11-7. (a) Lectura de datos de un archivo en el NFS versión 3. (b) Lectura de datos mediante un procedimiento compuesto en la versión 4.

Este método requería dos RPC sucesivas. La desventaja fue evidente cuando se consideró el uso del NFS en un sistema de área amplia. En ese caso, la latencia extra de una segunda RPC degradó el desempeño. Para evitar esos problemas, el NFSv4 soporta **procedimientos compuestos** mediante los cuales varias RPC pueden agruparse en una sola solicitud, como se muestra en la figura 11-7(b).

En nuestro ejemplo, el cliente combina la solicitud de búsqueda y lectura en una sola RPC. En el caso de la versión 4, también es necesario abrir el archivo antes de que pueda ocurrir la lectura. Después de que se ha buscado el manejador de archivo, se transfiere a la operación open, luego de lo cual el servidor continúa con la operación read. El efecto total en este ejemplo es que sólo se tienen que intercambiar dos mensajes entre el cliente y el servidor.

No existen semánticas transaccionales asociadas con los procedimientos compuestos. Las operaciones agrupadas en un procedimiento compuesto simplemente se manejan en el orden en que fueron solicitadas. Si existen operaciones concurrentes de otros clientes, entonces no se toman medidas para evitar conflictos. Si por cualquier razón falla una operación, entonces ya no se ejecutan más operaciones en el procedimiento compuesto, y los resultados encontrados hasta ese momento se regresan al cliente. Por ejemplo, si falla la operación lookup, ni siquiera se intenta una operación open subsiguiente.

11.3.2 El subsistema RPC2

Otra interesante mejora para RPC se desarrolló como una parte del sistema de archivo Coda (Kistler y Satyanarayanan, 1992). El **RPC2** es un paquete que ofrece RPC confiables encima del protocolo UDP (no confiable). Cada vez que un procedimiento remoto es invocado, el código de cliente RPC2 inicia un nuevo hilo que envía una solicitud de invocación al servidor y posteriormente se bloquea hasta que recibe una respuesta. Como el procesamiento de la solicitud puede llevarse un tiempo arbitrario para estar completo, el servidor regularmente devuelve mensajes al cliente para informarle que sigue trabajando en la solicitud. Si el servidor deja de funcionar, tarde o temprano el hilo advertirá que cesaron los mensajes y reportará la falla a la aplicación invocadora.

Un aspecto interesante del RPC2 es su soporte de efectos colaterales. Un **efecto colateral** es un mecanismo mediante el cual el cliente y el servidor pueden comunicarse usando un protocolo específico de una aplicación. Consideremos, por ejemplo, un cliente que abre un archivo en un servidor de videos. Lo que se requiere en este caso es que el cliente y el servidor establezcan un flujo de datos continuo con un modo de transmisión isócrono. En otros términos, se garantiza que la transferencia de datos del servidor al cliente se encuentra dentro de un retraso máximo y mínimo de extremo a extremo.

El RPC2 permite que el cliente y el servidor establezcan una conexión distinta para transferir a tiempo los datos de video al cliente. El establecimiento de la conexión ocurre como un efecto colateral de una RPC al servidor. Con esta finalidad, el sistema en tiempo de ejecución RPC2 proporciona una interfaz de rutinas de efecto colateral que tiene que ser implementada por el desarrollador de la aplicación. Por ejemplo, existen rutinas para establecer una conexión y rutinas para transferir datos. Estas rutinas son automáticamente invocadas por el sistema en tiempo de ejecución RPC2 en el cliente y el servidor, respectivamente, pero por otra parte su implementación es completamente independiente del RPC2. Este principio de efectos colaterales se muestra en la figura 11-8.

Otra característica del RPC2 que lo hace diferente de otros sistemas RPC es su soporte de multitransmisión. En Coda, un importante tema de diseño es que los servidores no pierden de vista

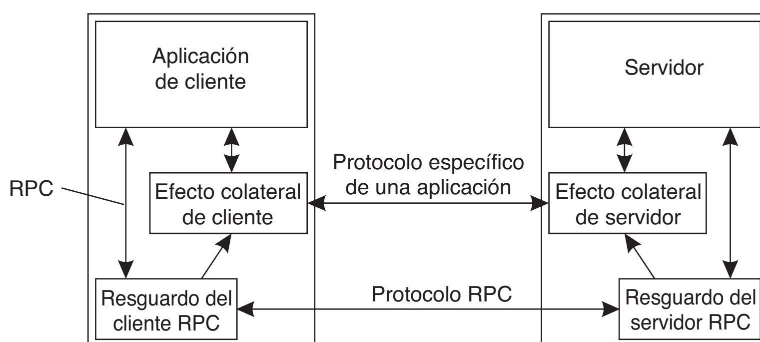


Figura 11-8. Efectos colaterales en el sistema RPC2 de Coda.

cuáles servidores tienen una copia local de un archivo. Cuando un archivo se modifica, un servidor invalida las copias locales al notificar a los clientes apropiados mediante una RPC. Claramente, si un servidor puede notificar a sólo un cliente a la vez, la invalidación de todos los clientes puede llevarse algo de tiempo, como se ilustra en la figura 11-9(a).

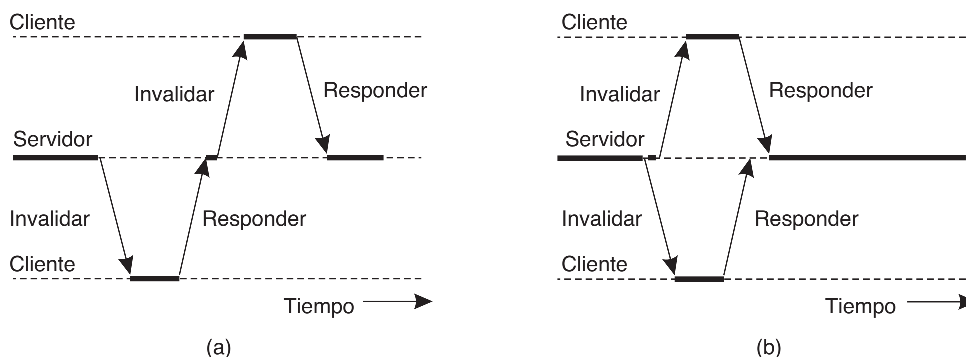


Figura 11-9. (a) Envío de mensajes de invalidación de uno en uno. (b) Envío de mensajes de invalidación en paralelo.

El problema es provocado por el hecho de que una RPC puede fallar de vez en cuando. La invalidación de los archivos en un estricto orden secuencial puede demorarse considerablemente porque el servidor no puede llegar a un cliente posiblemente congelado, pero se dará por vencido con ese cliente sólo después de un tiempo de expiración relativamente largo. Entre tanto, otros clientes continuarán leyendo en sus copias locales.

Una solución alternativa (y mejor) se muestra en la figura 11-9(b). Aquí, en lugar de invalidar cada copia una por una, el servidor envía un mensaje de invalidación a todos los clientes al mismo tiempo. Por consiguiente, todos los clientes que no fallaron son notificados en el mismo

tiempo que se llevaría realizar una RPC inmediata. También, el servidor advierte que dentro del tiempo de expiración usual ciertos clientes no responden a la RPC, y puede declararlos como congelados.

RPC paralelas se implementan por medio del sistema **MultiRPC**, el cual es parte del paquete RPC2 (Satyanarayanan y Siegel, 1990). Un importante aspecto del MultiRPC es que la invocación paralela de RPC es totalmente transparente para el que recibe la llamada. En otros términos, el receptor de una llamada MultiRPC no puede distinguirla de una RPC normal. Del lado de quien llama, la ejecución paralela también es muy transparente. Por ejemplo, la semántica del MultiRPC en la presencia de fallas es casi la misma de una RPC normal. Asimismo, los mecanismos de efecto colateral pueden ser utilizados como antes.

En esencia, el MultiRPC se implementa ejecutando múltiples RPC puestas en paralelo. Esto significa que quien llama envía explícitamente una solicitud RPC a cada receptor. Sin embargo, en lugar de ponerse a esperar de inmediato una respuesta, aplaza el bloqueo hasta que todas las solicitudes han sido enviadas. En otros términos, quien llama invoca varias RPC en una dirección, tras de lo cual se bloquea hasta que se reciben todas las respuestas de los receptores que no fallaron. Un método alternativo de la ejecución en paralelo de RPC en el MultiRPC se logra estableciendo un grupo de multitransmisión, y enviando una RPC a todos los miembros del grupo mediante multitransmisión IP.

11.3.3 Comunicación orientada a archivos en Plan 9

Por último, vale la pena mencionar un método completamente diferente de manejar la comunicación en sistemas de archivo distribuidos. El **Plan 9** (Pike y cols., 1995) no es tanto un sistema de archivo distribuido, sino más bien un *sistema distribuido basado en archivos*. Todos los recursos son accesados de igual modo, es decir, con sintaxis y operaciones como de archivo, incluidos recursos tales como procesos e interfaces de red. Esta idea se heredó de UNIX, el cual también intenta ofrecer interfaces como de archivo para comunicarse con los recursos, aunque se ha explotado de manera adicional y más consistentemente en Plan 9. Como ilustración, diremos que las interfaces de red están representadas por un sistema de archivo, en este caso compuesto a partir de un conjunto de archivos especiales. Este método es similar a UNIX, aunque en UNIX las interfaces de red están representadas por archivos y no por sistemas de archivo. (Observemos que en este contexto, un sistema de archivo de nuevo es un dispositivo de bloques lógicos que contiene todos los datos y metadatos comprendidos en un conjunto de archivos.) En Plan 9, por ejemplo, una conexión TCP individual está representada por un subdirectorio compuesto por los archivos mostrados en la figura 11-10.

El archivo *ctl* se utiliza para enviar comandos de control a la conexión. Por ejemplo, para abrir una sesión telnet para una máquina con dirección IP 192.31.231.42 utilizando el puerto 23, se requiere que el remitente escriba la cadena de texto “connect 192.31.231.42!23” en el archivo *ctl*. El destinatario previamente habría escrito la cadena “announce 23” en su propio archivo *ctl*, para indicar que puede aceptar solicitudes de sesión entrantes.

El archivo *data* se utiliza para intercambiar datos simplemente con realizar las operaciones read y write. Estas operaciones siguen la semántica UNIX usual para operaciones con archivos.

Archivo	Descripción
ctl	Utilizado para escribir comandos de control específicos del protocolo de escritura
datos	Utilizado para leer y escribir datos
escuchar	Utilizado para aceptar solicitudes de establecimiento de conexión entrantes
local	Proporciona información de la conexión del lado de quien llama
remoto	Proporciona información del otro lado de la conexión
estado	Proporciona información de diagnóstico sobre el estado actual de la conexión

Figura 11-10. Archivos asociados con una sola conexión TCP en Plan 9.

Por ejemplo, para escribir datos para una conexión, un proceso simplemente invoca la operación

```
res = write(fd, buf, nbytes);
```

donde *fd* es el descriptor de archivo regresado después de abrir el archivo de datos; *buf* es un apuntador de un bufer que contiene los datos a escribir, y *nbytes* es la cantidad de bytes que deberán ser extraídos del bufer. La cantidad de bytes escritos realmente es devuelta y guardada en la variable *res*.

El archivo *listen* se utiliza para esperar solicitudes de establecimiento de conexión. Después de que un proceso ha anunciado su deseo de aceptar nuevas conexiones, puede bloquear la operación read en el archivo *listen*. Si llega una solicitud, la llamada regresa un descriptor de archivo a un nuevo archivo *ctl* correspondiente a un directorio de conexión recién creado. Así vemos cómo se puede realizar un método orientado a archivos en relación con la comunicación.

11.4 ASIGNACIÓN DE NOMBRES

La asignación de nombres desempeña, indiscutiblemente, un rol importante en los sistemas de archivo distribuidos. En virtualmente todos los casos, los nombres están organizados en un espacio de nombre jerárquico como los espacios que estudiamos en el capítulo 5. A continuación consideraremos otra vez el NFS como representativo de cómo se maneja la asignación de nombres en sistemas de archivo distribuidos.

11.4.1 Asignación de nombres en NFS

La idea fundamental que constituye la base del modelo de asignación de nombres NFS es proporcionar a los clientes un acceso completamente transparente a un sistema de archivo remoto mantenido por un servidor. Esta transparencia se logra permitiendo que el cliente sea capaz de montar un sistema de archivo remoto en su propio sistema de archivo local, como se muestra en la figura 11-11.

En lugar de montar todo un sistema de archivo, el NFS permite que los clientes monten sólo una parte, como también se muestra en la figura 11-11. Se dice que un servidor **exporta** un directorio cuando pone a éste y a sus entradas a disposición de sus clientes. Un directorio exportado puede montarse en el espacio de nombre local del cliente.

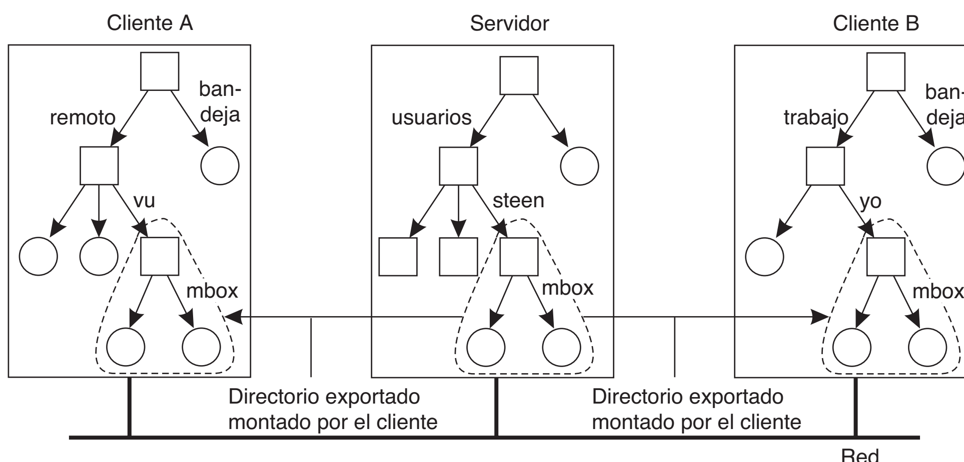


Figura 11-11. Montaje (de una parte) de un sistema de archivo remoto en NFS.

Este método de diseño tiene una seria implicación: en principio, los usuarios no comparten espacios de nombre. Como se muestra en la figura 11-11, el archivo de nombre */remote/vu/mbox* en el cliente *A* se llama */work/me/mbox* en el cliente *B*. Un nombre de archivo depende, por consiguiente, de cómo organizan los clientes su propio espacio de nombre local, y en dónde se monten los directorios exportados. La desventaja de usar este método en un sistema de archivo distribuido es que compartir archivos se vuelve mucho más difícil. Por ejemplo, Alicia no puede contarle a Bob acerca de un archivo utilizando el nombre que ella le asignó a dicho archivo, ya que ese nombre puede tener un significado enteramente distinto en el espacio de nombre de los archivos de Bob.

Existen varias formas de resolver este problema, pero la más común es proporcionar a cada cliente un espacio de nombre estandarizado en parte. Por ejemplo, cada cliente puede estar utilizando el directorio local */usr/bin* para montar un sistema de archivo que contiene un conjunto estándar de programas disponibles para todos. Asimismo, se puede utilizar el directorio */local* como estándar para montar un sistema de archivo local ubicado en el host del cliente.

Un servidor NFS, por sí mismo, puede montar directorios exportados por otros servidores. Sin embargo, no se permite que los exporte a sus propios clientes. En cambio, un cliente tendrá que montarlos explícitamente del servidor que los contiene, como se muestra en la figura 11-12. Esta restricción se deriva en parte de la simplicidad. Si un servidor pudiera exportar un directorio montado desde otro servidor, tendría que regresar manejadores de archivo especiales que incluyan un identificador para un servidor. El NFS no soporta esa clase de manejadores de archivo.

Para explicar este punto más a fondo, supongamos que el servidor *A* aloja un sistema de archivo FS_A del cual exporta el directorio */packages*. Este directorio contiene un subdirectorio */draw* que actúa como punto de montaje para un sistema de archivo FS_B exportado por el servidor *B* y montado por *A*. Se permite que *A* también exporte */packages/draw* a sus propios clientes, y se

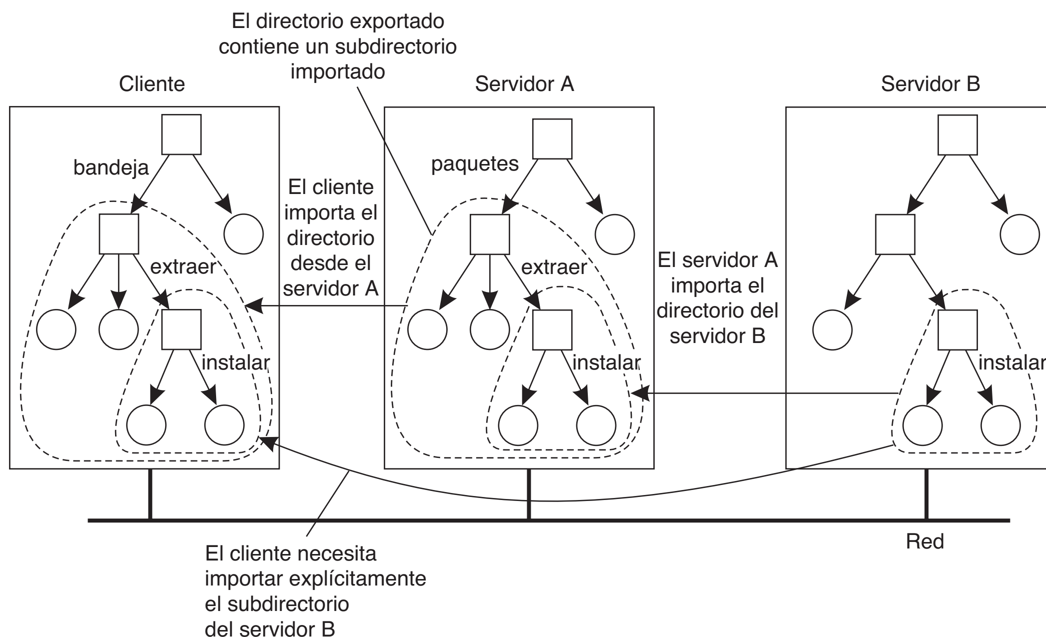


Figura 11-12. Montaje de directorios anidados tomados de múltiples servidores en NFS.

asume que un cliente ha montado */packages* en su propio directorio */bin* como se muestra en la figura 11-12.

Si la resolución del nombre es iterativa (como en el caso del NFSv3), entonces para resolver el nombre */bin/draw/install*, el cliente se pone en contacto con el servidor *A* cuando localmente ha resuelto */bin* y solicita a *A* que regrese un manejador de archivo para el directorio */draw*. En ese caso, el servidor *A* deberá regresar un manejador de archivo que incluya un identificador para el servidor *B*, para que sólo *B* pueda resolver el resto del nombre de ruta, en este caso */install*. Como ya se dijo, el NFS no soporta esta clase de resolución de nombre.

La resolución de nombre en NFSv3 (y versiones anteriores) es estrictamente iterativa en el sentido de que solamente puede ser buscado un solo nombre de archivo a la vez. En otros términos, la resolución de un nombre tal como */bin/draw/install* requiere tres llamadas distintas al servidor NFS. Además, el cliente es totalmente responsable de implementar la resolución de un nombre de ruta. El NFSv4 también soporta búsquedas repetitivas. En este caso, un cliente puede transferir todo un nombre de ruta a un servidor y pedirle que lo resuelva.

Existe otra peculiaridad con las búsquedas de nombres NFS que se resolvieron con la versión 4. Consideremos un servidor de archivos que aloja varios sistemas de archivo. Con la resolución de nombre estricta iterativa de la versión 3, siempre que se buscaba un directorio donde otro sistema de archivo estaba montado, la búsqueda podía regresar el manejador de archivo del directorio. Posteriormente, leyendo el directorio podía regresar su contenido *original*, no el contenido del directorio raíz del sistema de archivo montado.

Para explicar esto, supongamos que en el capítulo previo ambos sistemas de archivo FS_A y FS_B están alojados en el mismo servidor. Si el cliente montó */packages* en su directorio local */bin*, entonces al buscar el nombre de archivo *draw* en el servidor regresaría su manejador *draw*. Una llamada subsiguiente al servidor solicitándole una lista de entradas en el directorio *draw* por medio de readdir regresaría entonces la lista de entradas realizadas en el directorio que *originalmente* se guardaron en FS_A en el subdirectorio */packages/draw*. Sólo si el cliente hubiera montado también el sistema de archivo FS_B, sería posible resolver apropiadamente el nombre de ruta *draw/install* con respecto a */bin*.

NFSv4 soluciona este problema al permitir búsquedas de puntos de montaje entrecruzados en un servidor. En particular, lookup regresa el manejador de archivo del directorio *montado* en lugar del directorio original. El cliente puede detectar que la búsqueda cruzó un punto de montaje inspeccionando el identificador del sistema de archivo del archivo buscado. Si se requiere, el cliente puede montar también localmente el sistema de archivo.

Manejadores de archivo

Un manejador de archivo es una referencia a un archivo localizado dentro de un sistema de archivo. Es independiente del nombre del archivo al que se refiere. El servidor que aloja el sistema de archivo crea un manejador de archivo único con respecto a todos los sistemas exportados por él. Se genera en el momento en que el archivo es creado. El cliente permanece ignorante del contenido de un manejador de archivo; es completamente opaco. En el NFS versión 2 los manejadores de archivo eran de 32 bytes, pero en la versión 3 eran variables hasta 64 bytes, y de 128 bytes en la versión 4. Por supuesto, la longitud de un manejador de archivo no es opaca.

De modo ideal, un manejador de archivo es implementado como un verdadero identificador para un archivo con respecto a un sistema de archivo. Entre otras cosas, esto significa que en tanto exista el archivo, deberá tener uno y el mismo manejador. Este requerimiento de persistencia permite que un cliente guarde un manejador de archivo una vez que el archivo asociado ha sido buscado por medio de su nombre. Un beneficio es el desempeño: como la mayoría de las operaciones con archivos requiere un manejador de archivo en lugar de un nombre, el cliente puede evitar tener que buscar un nombre repetidamente antes de realizar cada operación. Otro beneficio de este método es que así el cliente puede acceder al archivo independientemente de sus nombres (actuales).

Como un manejador de archivo puede ser localmente guardado por un cliente, también es importante que un servidor no reutilice un manejador después de borrar un archivo. De lo contrario, un cliente puede acceder equivocadamente al archivo erróneo cuando utilice su manejador de archivo localmente guardado.

Observe que la combinación de búsquedas de nombre iterativas y no permitiendo que una operación de búsqueda cruce un punto de montaje presenta un problema con la obtención de una manejador de archivo inicial. Para acceder a archivos localizados en un sistema de archivo remoto, un cliente deberá proporcionar al servidor un manejador del archivo si el directorio donde tendrán lugar las búsquedas, junto con el nombre del archivo o directorio a resolver. NFSv3 resuelve este problema aplicando un protocolo de montaje por separado, mediante el cual un cliente monta en realidad un sistema de archivo remoto. Después del montaje, el cliente es transferido de vuelta al **manejador de archivo raíz** del sistema de archivo montado, el cual se utiliza posteriormente como punto de inicio para buscar nombres.

En el NFSv4, este problema se resuelve con una operación putrootfh para indicarle al servidor que resuelva todos los nombres de archivo con respecto al manejador de archivo raíz del sistema de archivo que gestiona. Se puede utilizar el manejador de archivo raíz para buscar cualquier otro manejador de archivo en el sistema de archivo del servidor. Este método tiene el beneficio adicional de que no se requiere un protocolo de montaje distinto. En cambio, el montaje puede incorporarse al protocolo regular para buscar archivos. Un cliente simplemente puede montar un sistema de archivo solicitando al servidor que resuelva nombres con respecto al archivo raíz del sistema de archivo por medio de putrootfh.

Automontaje

Como se mencionó, el modelo de asignación de nombre NFS esencialmente proporciona a los usuarios su propio espacio de nombre. En este modelo, compartir puede volverse difícil si los usuarios nombran el mismo archivo en forma diferente. Una solución a este problema es proporcionar a cada usuario un espacio de nombre local parcialmente estandarizado, y luego montar sistemas de archivo remotos de igual modo para cada usuario.

Otro problema con el modelo de asignación de nombres NFS tiene que ver con decidir *cuándo* deberá ser montado un sistema de archivo remoto. Consideremos un gran sistema de miles de usuarios; suponiendo que cada usuario tiene un directorio local */home* utilizado para montar los directorios de inicio de otros usuarios. Por ejemplo, el directorio de inicio de Alicia puede estar localmente disponible para ella como */home/alicia*, aunque los archivos en sí estén guardados en un servidor remoto. Este directorio puede ser montado automáticamente cuando Alicia inicie una sesión en su estación de trabajo. Además, Alicia puede tener acceso a los archivos públicos de Bob accediendo al directorio de éste mediante */home/bob*.

La pregunta, sin embargo, es si el directorio de Bob también deberá ser montado automáticamente cuando Alicia inicie una sesión. El beneficio de este método sería que todo el negocio de montar sistemas de archivo fuese transparente para Alicia. Sin embargo, de seguir esta política para cada usuario, el inicio de sesión implicaría una enorme sobrecarga de comunicación y administración. Además, requeriría conocer de antemano a todos los usuarios. Un método mejor es montar en forma transparente otro directorio de inicio de usuario conforme a la demanda, es decir, cuando se requiera primero.

En NFS, un **automontador** se encarga del montaje conforme a la demanda de un sistema de archivo remoto (o en realidad un directorio exportado), el cual funciona como un proceso distinto en la máquina del cliente. El principio que constituye la base de un automontador es relativamente simple. Consideremos un automontador simple implementado como un servidor NFS al nivel de usuario sobre un sistema operativo UNIX. Para implementaciones alternativas, vea Callaghan (2000).

Suponga que para cada usuario, los directorios de inicio de todos los usarios están disponibles a través de un directorio local */home*, tal como ya se describió. Cuando una máquina cliente se inicia, el automontador empieza el montaje de este directorio. El efecto de este montaje local es que siempre que un programa intente acceder a */home*, el kernel de UNIX emitirá una operación lookup al cliente NFS, quien en este caso remitirá la solicitud al automontador en su rol de servidor NFS, como se muestra en la figura 11-13.

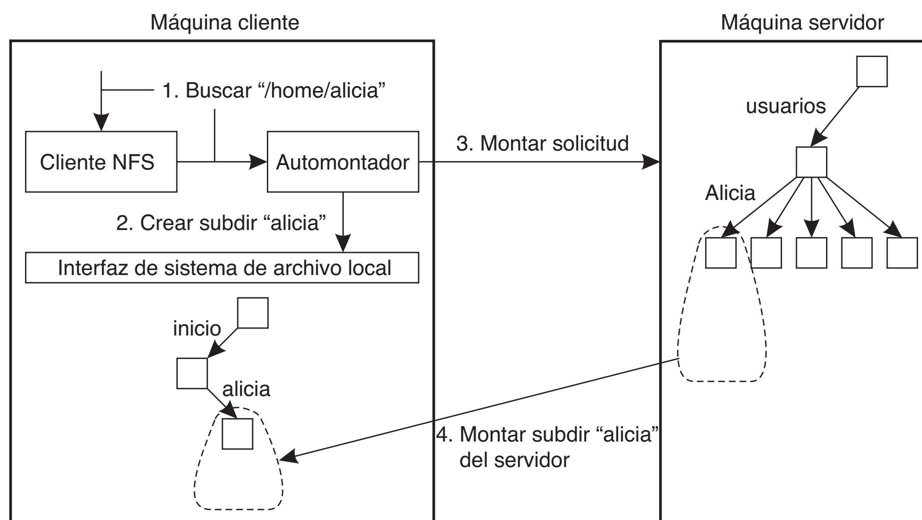


Figura 11-13. Automontador simple para NFS.

Por ejemplo, supongamos que Alicia inicia una sesión. El programa de inicio de sesión intenta leer el directorio */home/alicia* para buscar información tal como guiones de inicio de sesión. El automontador recibirá, por tanto, la solicitud de examinar el subdirectorio */home/alicia*, razón por la cual primero creará un subdirectorio */alicia* in*/home*. Luego examinará el servidor NFS que exporta el directorio de inicio de Alicia para, posteriormente, montar dicho directorio en */home/alicia*. En ese punto, el programa de inicio puede proseguir.

El problema con este método es que el automontador deberá intervenir en todas las operaciones de archivo para garantizar la transparencia. Si un archivo referido no está localmente disponible porque aún no se ha montado el sistema de archivo correspondiente, el automontador tendrá que saberlo. En particular, deberá manejar todas las solicitudes de lectura y escritura, incluso para sistemas de archivo que ya hayan sido montados. Este método puede incurrir en un gran problema de desempeño. Sería mejor tener automontadores sólo para directorios de montaje y desmontaje, pero de otro modo se mantiene fuera del bucle.

Una solución simple es permitir que el automontador monte directorios en un subdirectorio especial e instale un vínculo simbólico para cada directorio montado. Este método se muestra en la figura 11-14.

En nuestro ejemplo, los directorios de inicio de usuario se montan como subdirectorios de */tmp_mnt*. Cuando Alicia inicia una sesión, el automontador monta su directorio de inicio en */tmp_mnt/home/alicia* y crea un vínculo simbólico */home/alicia* que se refiere a dicho subdirectorio. En este caso, siempre que Alicia ejecuta un comando tal como

```
ls–l/home/alicia
```

el servidor NFS que exporta el directorio de inicio de Alicia es contactado directamente sin que intervenga el automontador.

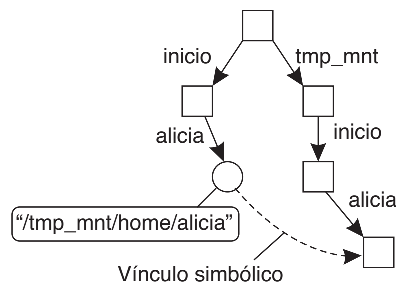


Figura 11-14. Utilización de vínculos simbólicos con automontaje.

11.4.2 Construcción de un gran sistema de nombres global

Los grandes sistemas distribuidos comúnmente se construyen aglutinando varios sistemas heredados en uno solo. Cuando se trata de ofrecer acceso compartido a archivos, tener un espacio de nombre global es casi la aglutinación mínima que se desearía tener. En la actualidad, los sistemas de archivo se abren principalmente para compartir mediante el uso de medios primitivos tales como el acceso a través de FTP. Este método, por ejemplo, se utiliza a menudo en cómputo de Malla (Grids).

Los sistemas de archivo distribuidos verdaderamente de área amplia siguen métodos más complejos, aunque a menudo requieren modificaciones en el kernel de cada sistema operativo para ser adoptados. Por consiguiente, los investigadores han estado buscando métodos para integrar sistemas de archivo existentes en un solo espacio de nombre global, pero utilizando soluciones sólo a nivel del usuario. Un sistema de ese tipo, llamado simplemente **servicio global de nombre de espacio** (**GNS**, por sus siglas en inglés), es propuesto por Anderson y colaboradores (2004).

El GNS no proporciona interfaces para acceder a archivos. En cambio, simplemente proporciona los medios para establecer un espacio de nombre global donde varios espacios de nombre existentes se han fusionado. Con esta finalidad, un cliente GNS mantiene un árbol virtual donde cada nodo es o un directorio o una **unión**. Una unión es un nodo especial que indica que otro proceso se encargó de la resolución de nombre, y como tal guarda cierto parecido con un punto de montaje en un sistema de archivo tradicional. Existen cinco tipos diferentes de uniones, como se muestra en la figura 11-15.

Una unión GNS simplemente se refiere a otra instancia GNS, la cual es simplemente otro árbol virtual alojado en posiblemente otro proceso. Las dos uniones lógicas contienen la información requerida para contactar un servicio de localización. La segunda proporcionará la dirección de contacto para acceder a un sistema de archivo y a un archivo, respectivamente. Un nombre de sistema de archivo físico se refiere a un sistema de archivo localizado en otro servidor, y corresponde en gran medida a una dirección de contacto que una unión lógica necesitaría. Por ejemplo, una URL tal como *ftp://ftp.cs.vu.nl/pub* contendría información para acceder a archivos en el servidor FTP indicado. En forma similar, una URL tal como *http//www.cs.vu.nl/index.htm* es un ejemplo típico de un nombre de archivo físico.

Unión	Descripción
Unión GNS	Se refiere a otra instancia GNS
Nombre de sistema de archivo lógico	Referencia a un subárbol donde se deberá buscar en un servicio de localización
Nombre de archivo lógico	Referencia a un archivo que deberá ser buscado en un servicio de localización
Nombre de sistema de archivo físico	Referencia a un subárbol remoto directamente accesible
Nombre de archivo físico	Referencia a un archivo remoto directamente accesible

Figura 11-15. Uniones en GNS.

Desde luego, una unión deberá contener toda la información requerida para continuar la resolución de nombre. Hay muchas formas de hacerlo, pero considerando la existencia de tantos sistemas de archivo diferentes, cada unión específica demandará su propia implementación. Afortunadamente, existen también formas comunes de acceder a archivos remotos, incluidos protocolos de comunicación con servidores NFS, servidores FTP y máquinas basadas en Windows (principalmente CIFS).

GNS tiene la ventaja de desacoplar la asignación de nombres a partir de su ubicación existente. De ninguna manera un árbol virtual tiene que ver con la ubicación física de archivos y directorios. Además, utilizando un servicio de localización también es posible mover archivos de un lado a otro sin que sus nombres se vuelvan irresolubles. En ese caso, la nueva ubicación física tiene que ser registrada en el servicio de localización. Observe que esto es lo mismo que presentamos en el capítulo 5.

11.5 SINCRONIZACIÓN

Prosigamos nuestro análisis enfocándonos en los temas de sincronización relacionados con los sistemas de archivo distribuidos. Existen varios aspectos que requieren atención. En primer lugar, implementar la sincronización en los sistemas de archivo no sería un problema si los archivos no fueran compartidos. Sin embargo, en un sistema distribuido, la semántica de archivos compartidos se vuelve un poco engañosa cuando el desempeño está en peligro. Con esta finalidad, se han propuesto diferentes soluciones de las cuales analizamos las más importantes a continuación.

11.5.1 Semántica de archivos compartidos

Cuando dos o más usuarios comparten el mismo archivo al mismo tiempo, es necesario definir con precisión la semántica de lectura y escritura para evitar problemas. En sistemas de un solo procesador que permiten a los procesos compartir archivos, tal como UNIX, la semántica establece normalmente que una operación read sigue a una operación write, la operación read regresa el valor que se acaba de escribir, como se muestra en la figura 11-16(a). Asimismo, ocurren dos operaciones write en rápida sucesión, seguidas por una operación read, el valor leído es el valor guardado por la última operación de escritura. De hecho, el sistema hace que se cumpla un ordenamiento en tiempo

absoluto en todas las operaciones y siempre regresa el valor más reciente. Haremos referencia a este modelo como **semántica UNIX**. Este modelo es fácil de entender y de implementar.

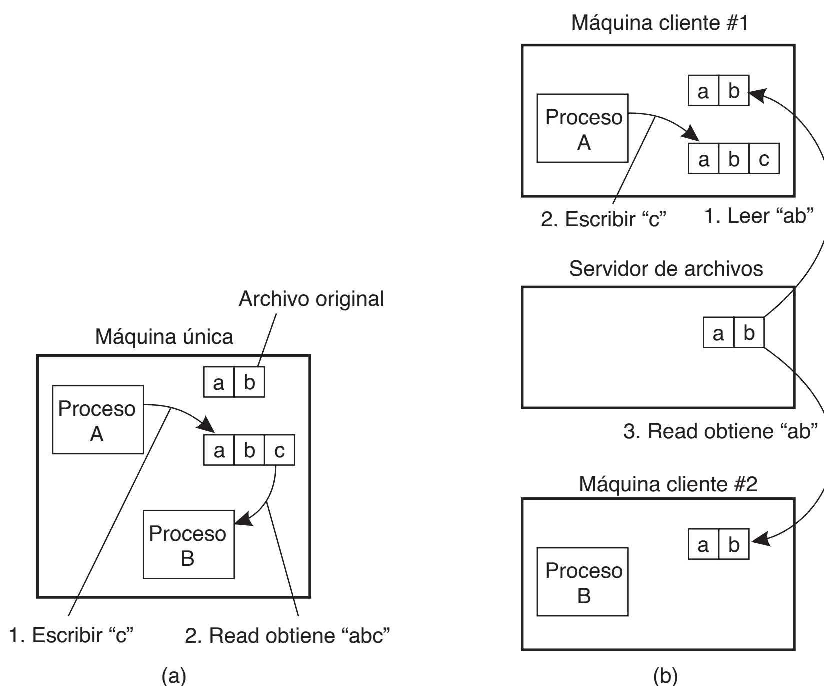


Figura 11-16. (a) En un solo procesador, cuando read sigue después de write, el valor regresado por la operación read es el valor que se acaba de escribir. (b) En un sistema distribuido con almacenamiento en la memoria caché, los valores obsoletos pueden ser regresados.

En un sistema distribuido, la semántica UNIX es fácil de lograr en tanto exista sólo un servidor y los clientes no guarden los archivos en la memoria caché. Todas las operaciones read y write se van directamente al servidor de archivos, el cual las procesa estrictamente en secuencia. Este método de semántica UNIX (excepto por el problema mínimo de que las demoras pueden provocar que una read ocurrida un microsegundo después de una write llegue primero al servidor, y que, por tanto, se obtenga el valor viejo).

En la práctica, sin embargo, el desempeño de un sistema distribuido en el cual todas las solicitudes de archivo deben dirigirse a un solo servidor con frecuencia es deficiente. Este problema a menudo se resuelve permitiendo que los clientes mantengan copias locales de archivos muy utilizados en sus cachés privados (locales). Aunque presentamos los detalles del almacenamiento de archivos en la memoria caché a continuación, por el momento es suficiente con señalar que si un cliente modifica localmente un archivo guardado en la memoria caché y poco después otro cliente lo lee desde el servidor, el segundo cliente obtendrá un archivo obsoleto, como se ilustra en la figura 11-16(b).

Una forma de superar esta dificultad es regresar al servidor todos los cambios a archivos guardados en la memoria caché de inmediato. Aunque este método es conceptualmente simple, en la práctica resulta ineficiente. Una solución alternativa es hacer más flexible la semántica del compartimiento de archivos. En lugar de requerir que una operación read vea los efectos de todas las operaciones previas write, se puede establecer una regla que diga: "Los cambios a un archivo abierto inicialmente son visibles sólo para el proceso (o la máquina) que lo modificó. Solamente cuando se cierre el archivo los cambios serán visibles para otros procesos (u otras máquinas)." La adopción de esta regla no cambia lo que sucede en la figura 11-16(b), pero sí redefine el comportamiento real (*B* obtiene el valor original del archivo) como el correcto. Cuando *A* cierra el archivo, envía una copia al servidor para que operaciones read subsiguientes obtengan el nuevo valor, como se requiere.

Esta regla es ampliamente implementada y se conoce como **semántica de sesión**. La mayoría de los sistemas distribuidos implementan semántica de sesión. Esto significa que, aunque en teoría siguen el modelo de acceso remoto de la figura 11-1(a), la mayoría de las implementaciones utilizan cachés locales, que efectivamente implementan el modelo de carga y descarga de la figura 11-1(b).

Con la semántica de sesión surge la pregunta acerca de qué sucede si dos o más clientes, al mismo tiempo, están guardando en el caché y modificando el mismo archivo. Una solución es decir que puesto que cada archivo se cierra en su oportunidad, su valor es regresado al servidor, por ello el resultado final depende de qué solicitud de cierre es la más recientemente procesada por el servidor. Una solución menos agradable, aunque más fácil de implementar, es decir que el resultado final es uno de los candidatos, pero dejar la alternativa de cuál sin especificar.

Un método completamente diferente para la semántica de compartimiento de archivos en un sistema distribuido es hacer que todos los archivos sean inmutables. No existe, por tanto, ninguna forma de abrir un archivo para escribir. En realidad, las únicas operaciones en archivos son create y read.

Lo que sí es posible es crear un archivo enteramente nuevo e ingresarlo al sistema de directorio con el nombre de un archivo previo existente, que entonces se vuelve inaccesible (por lo menos con ese nombre). Por tanto, aunque llega a ser imposible modificar el archivo *x*, sigue siendo posible reemplazar atómicamente *x* con uno nuevo. En otros términos, aunque los *archivos* no pueden ser actualizados, los *directorios* sí. Una vez decidido que los archivos no pueden ser cambiados en absoluto, el problema de cómo manejar dos procesos, uno de los cuales está escribiendo en un archivo y el otro está leyéndolo, simplemente desaparece y el diseño se simplifica en gran medida.

Lo que queda es el problema de qué sucede cuando dos procesos tratan de reemplazar el mismo archivo al mismo tiempo. Igual que con la semántica de sesión, la mejor solución parece ser permitir que uno de los archivos nuevos reemplace al viejo, o al último o no determinísticamente.

Un problema un tanto más difícil es qué hacer si un archivo es reemplazado mientras otro proceso está ocupado leyéndolo. Una solución es disponer las cosas de algún modo para que el lector continúe utilizando el archivo viejo, aun cuando ya no se encuentre en el directorio, en forma similar al modo en que UNIX permite a un proceso que tiene un archivo abierto continuar utilizándolo, incluso después de que ha sido borrado de todos los directorios. Otra solución es detectar que el archivo ha cambiado y hacer que fallen intentos subsiguientes de leerlo.

Una cuarta forma de ocuparse de los archivos compartidos en un sistema distribuido es utilizar transacciones atómicas. Resumiendo brevemente, para acceder a un archivo o grupo de archivos, un proceso ejecuta primero algún tipo de primitivo BEGIN_TRANSACTION para señalizar que lo que sigue debe ser ejecutado indivisiblemente. Entonces el sistema llama para leer y escribir uno o más archivos. Cuando el trabajo solicitado ha sido completado, se ejecuta un primitivo END_TRANSACTION. La propiedad fundamental de este método es que el sistema garantiza que todas las llamadas contenidas dentro de la transacción serán realizadas en orden, sin ninguna interferencia de otras transacciones concurrentes. Si dos o más transacciones se inician al mismo tiempo, el sistema garantiza que el resultado final será el mismo como si se estuvieran ejecutando en algún orden secuencial (no definido).

En la figura 11-17 se resumen los métodos que acabamos de analizar para el manejo de archivos compartidos en un sistema distribuido.

Método	Comentario
Semántica UNIX	Toda operación realizada en un archivo es visible al instante para todos los procesos
Semántica de sesión	Ningún cambio es visible para otros procesos hasta que el archivo se cierra
Archivos inmutables	No son posibles actualizaciones; simplifica el compartimentar y la replicación
Transacciones	Todos los cambios ocurren atómicamente

Figura 11-17. Cuatro formas de manejar archivos compartidos en un sistema distribuido.

11.5.2 Bloqueo de archivos

En arquitecturas de cliente-servidor con servidores sin estado, principalmente, se cuenta con medios adicionales para sincronizar el acceso a archivos compartidos. La forma tradicional de hacerlo es utilizar un gestor de bloqueo. Sin excepción, un gestor de bloqueo sigue el esquema de bloqueo centralizado que analizamos en el capítulo 6.

Sin embargo, las cosas no son tan simples como se acaba de describir. Aunque en general se despliega un gestor de bloqueo central, la complejidad del bloqueo se deriva de la necesidad de permitir el acceso concurrente al mismo archivo. Por esta razón, existe un gran número de bloqueos diferentes, y además, la granularidad de los bloqueos también puede diferir. Consideremos otra vez el NFSv4.

En su concepto, el bloqueo de archivos en NFSv4 es muy simple. Existen esencialmente sólo cuatro operaciones relacionadas con el bloqueo, como se muestra en la figura 11-18. El NFSv4 distingue los bloqueos de lectura de los bloqueos de escritura. Múltiples clientes pueden acceder al mismo tiempo a la misma parte de un archivo siempre que sólo lean datos. Se requiere un bloqueo de escritura para obtener acceso exclusivo para modificar una parte de un archivo.

La operación lock se utiliza para solicitar un bloqueo de lectura o escritura a lo largo de un intervalo consecutivo de bytes en un archivo. Esto es una operación de no bloqueo; si el bloqueo

Operación	Descripción
Lock	Crea un bloqueo en un rango de bytes
Lockt	Comprueba si se otorgó un bloqueo conflictivo
Locku	Retira un bloqueo a partir de un rango de bytes
Renew	Renueva el contrato de un bloqueo especificado

Figura 11-18. Operaciones en el NFSv4 relacionadas con el bloqueo de archivos.

no puede ser garantizado a causa de otro bloqueo conflictivo, el cliente obtiene un mensaje de error y tiene que sondear al servidor posteriormente. No existe un reintento automático. Alternativamente, el cliente puede solicitar al servidor que lo ponga en una lista en orden FIFO. En cuanto el bloqueo conflictivo desaparece, el servidor otorgará el siguiente bloqueo al cliente que se encuentra en la parte superior de la lista, siempre que el cliente sondee al servidor antes de que expire cierto tiempo. Este método evita que el servidor tenga que notificar a los clientes, y es justo con los clientes cuya solicitud de bloqueo no pudo ser otorgada porque las concesiones se hacen en orden FIFO.

La operación lockt se utiliza para comprobar si existe un bloqueo conflictivo. Por ejemplo, un cliente puede comprobar si se otorgaron bloqueos de lectura a lo largo de un intervalo específico de bytes en un archivo, antes de solicitar un bloqueo de escritura de dichos bytes. En el caso de que exista un conflicto, el cliente solicitante es informado con exactitud sobre quién lo está provocando y en qué intervalo de bytes. Esto puede ser implementado más eficientemente que lock, porque no existe la necesidad de abrir un archivo.

La eliminación de un bloqueo de un archivo se realiza por medio de la operación locku.

Se otorgan bloqueos durante un tiempo específico (determinado por el servidor). En otros términos, tienen un contrato asociado. A menos que un cliente renueve el contrato sobre un bloqueo otorgado, el servidor automáticamente lo eliminará. Este método también se sigue para otros servicios provistos por un servidor y ayuda en la recuperación después de fallas. Con la operación renew, un cliente solicita al servidor que renueve el contrato de su bloqueo (y, de hecho, también de otros recursos).

Además de estas operaciones, también existe una forma implícita de bloquear un archivo, conocida como **compartimiento de reservación**. El compartimiento de reservación es totalmente independiente del bloqueo y puede ser utilizado para implementar el NFS en sistemas basados en Windows. Cuando un cliente abre un archivo, especifica el tipo de acceso que requiere (es decir, *LECTURA*, *ESCRITURA* o *AMBOS*) y qué tipo de acceso deberá negar el servidor a otros clientes (*NINGUNO*, *LECTURA*, *ESCRITURA* o *AMBOS*). Si el servidor no puede satisfacer los requerimientos del cliente, la operación open fallará para ese cliente. En la figura 11-19 se muestra con exactitud lo que sucede cuando un nuevo cliente abre un archivo que ya ha sido abierto con éxito por otro cliente. Para un archivo que ya está abierto, distinguimos dos variables de estado diferentes. El estado de acceso especifica cómo está siendo accesado actualmente el archivo por el cliente actual. El estado de negación especifica qué accesos no le están permitidos a clientes nuevos.

En la figura 11-19(a), mostramos lo que sucede cuando un cliente trata de abrir un archivo para solicitar un tipo específico de acceso, dado el estado de negación actual de dicho archivo.

Estado de denegación de archivo actual

Solicitud de acceso	NINGUNA	LECTURA	ESCRITURA	AMBAS
LECTURA	Tuvo éxito	Falló	Tuvo éxito	Falló
ESCRITURA	Tuvo éxito	Tuvo éxito	Falló	Falló
AMBAS	Tuvo éxito	Falló	Falló	Falló

(a)

Estado de denegación de archivo solicitado

Estado de acceso actual	NINGUNA	LECTURA	ESCRITURA	AMBAS
LECTURA	Tuvo éxito	Falló	Tuvo éxito	Falló
ESCRITURA	Tuvo éxito	Tuvo éxito	Falló	Falló
AMBAS	Tuvo éxito	Falló	Falló	Falló

(b)

Figura 11-19. Resultado de una operación abierta con reservaciones compartidas en NFS. (a) Cuando el cliente solicita acceso compartido dado el estado de denegación actual. (b) Cuando el cliente solicita un estado de denegación dado el estado de acceso actual al archivo.

Asimismo, la figura 11-19(b) muestra el resultado de abrir un archivo que actualmente está siendo accesado por otro cliente, pero ahora para solicitar que ciertos tipos de acceso sean deshabilitados.

El NFSv4 de ningún modo es la excepción cuando se trata de ofrecer mecanismos de sincronización para archivos compartidos. De hecho, por ahora se acepta que cualquier conjunto simple de primitivas tal como sólo el bloqueo total de archivos refleja un diseño deficiente. En los esquemas de bloqueo, la complejidad se deriva principalmente de que se requiere una granularidad fina del bloqueo para permitir el acceso concurrente a archivos compartidos. Se ha intentado reducir la complejidad al mismo tiempo que se mantiene el desempeño [vea, por ejemplo, Burns y cols. (2001)], pero la situación continúa siendo un tanto insatisfactoria. Al final, es posible que pensemos en rediseñar por completo nuestras aplicaciones para escalabilidad en lugar de tratar de parchar situaciones derivadas del deseo de compartir datos como lo hicimos en sistemas no distribuidos.

11.5.3 Compartimiento de archivos en Coda

En NFS, las semánticas de sesión dictan que el último proceso que cierre un archivo propagará sus cambios al servidor; se perderá cualquier actualización realizada en sesiones previas concurrentes. Un método un tanto más sutil también puede ser aplicado. Para acomodar la compartimentación de archivos, el sistema de archivo Coda (Kistler y Satyanaryanan, 1992) utiliza un esquema de asignación especial que guarda ciertas similitudes con las reservaciones de compartimiento en NFS. Para entender cómo funciona el esquema, lo siguiente es muy importante. Cuando un cliente abre con éxito un archivo *f*, una copia completa de éste se transfiere a la máquina del cliente. El servidor

registra que el cliente tiene una copia de *f*. Hasta aquí, este método es similar a la delegación de apertura incluida en NFS.

Supongamos ahora que el cliente *A* abrió el archivo *f* para escritura. Cuando otro cliente *B* también desee abrirlo, el archivo fallará. Esta falla es provocada porque el servidor registró que el cliente *A* pudiera ya haber modificado el archivo *f*. Por otra parte, si el cliente *A* hubiera abierto *f* para lectura, el intento de un cliente *B* de obtener una copia del servidor para lectura tendría éxito. También tendría éxito el intento de *B* de abrir el archivo para escritura.

Ahora consideremos lo que sucede cuando varias copias de *f* se guardaron localmente en varios clientes. Dado lo que acabamos de afirmar, sólo un cliente será capaz de modificar *f*. Si este cliente lo modifica y posteriormente lo cierra, el archivo será transferido de vuelta al servidor. Sin embargo, cualquier cliente puede proseguir leyendo su copia local a pesar de que realmente no esté actualizada.

La razón de este comportamiento en apariencia inconsistente es que una sesión se trata como una transacción en Coda. Consideremos la figura 11-20, la cual muestra la línea de tiempo de dos procesos *A* y *B*. Supongamos que *A* abrió *f* para leerlo, ello conduce a la sesión S_A. El cliente *B* lo abrió para escritura, mostrado como sesión S_B.

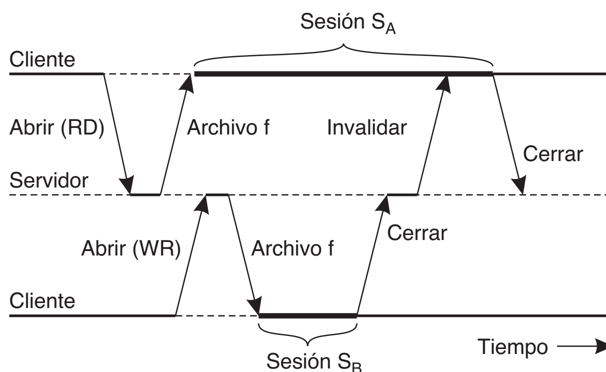


Figura 11-20. Comportamiento transaccional al compartir archivos en Coda.

Cuando *B* cierra la sesión S_B, transfiere la versión actualizada de *f* al servidor, el cual enviará entonces un mensaje de invalidación a *A*. *A* sabrá así que está leyendo un versión vieja de *f*. Sin embargo, desde un punto de vista transaccional, esto realmente no importa porque se podría considerar que la sesión S_A se programó antes que la sesión S_B.

11.6 CONSISTENCIA Y REPLICACIÓN

El almacenamiento en la memoria caché y la replicación desempeñan un rol muy importante en sistemas de archivo distribuidos, más notoriamente cuando están diseñados para que operen en redes de área amplia. En lo que sigue, examinaremos varios aspectos relacionados con el almacenamiento

en la memoria caché del lado del cliente del archivo de datos, así como la replicación de servidores de archivo. También, analizamos el rol de la replicación en el sistema de compartimiento de archivos punto a punto.

11.6.1 Almacenamiento en la memoria caché del lado del cliente

Para ver cómo se realiza el almacenamiento en la memoria caché del lado del cliente en la práctica, regresemos a los sistemas ejemplo NFS y Coda.

Almacenamiento en la memoria caché en NFS

El almacenamiento en la memoria caché en NFSv3 se dejó principalmente fuera del protocolo. Este método conduce a la implementación de diferentes políticas de almacenamiento en la memoria caché, la mayoría de las cuales nunca garantizan consistencia. En el mejor de los casos, los datos guardados en la memoria caché podrían estar alojados por algunos segundos, en comparación con la seguridad que se tiene en un servidor. Sin embargo, esta implementación permite que los datos se almacenen en la memoria caché durante 30 segundos sin que el cliente lo sepa. Este estado de cosas es menos que deseable.

El NFSv4 resuelve algunos de estos problemas de consistencia, pero esencialmente continúa dejando que la consistencia de la memoria caché sea manejada en una forma dependiente de una implementación. El modelo de almacenamiento en la memoria caché asumido por el NFS se muestra en la figura 11-21. Cada cliente puede tener un caché de memoria que contenga datos previamente leídos del servidor. Además, también puede haber un caché de disco agregado como extensión de la memoria caché, utilizando los mismos parámetros de consistencia.

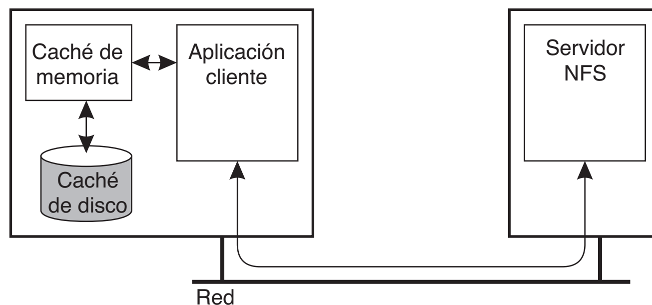


Figura 11-21. Almacenamiento en caché del lado del cliente en NFS.

Por lo común, los clientes guardan en caché datos de archivos, atributos, manejadores de archivo y directorios. Existen diferentes estrategias para manejar la consistencia de los datos guardados en caché, los atributos guardados en caché y así sucesivamente. Primero daremos un vistazo al almacenamiento de datos en el caché de datos de archivo.

El NFSv4 soporta dos métodos diferentes de guardar datos de archivo en caché. El más simple es cuando un cliente abre un archivo y guarda en caché los datos que obtiene del servidor como resultado de varias operaciones read. Además, las operaciones write pueden ser realizadas también en caché. Cuando un cliente cierra el archivo, el NFS requiere que si las modificaciones se han

llevado a cabo, los datos guardados en caché sean devueltos de inmediato al servidor. Este método corresponde a la implementación de semánticas de sesión tal como se vio con anterioridad.

Una vez que (parte de) un archivo ha sido guardado en la memoria caché, un cliente puede conservar ahí sus datos incluso después de cerrar el archivo. También, varios clientes localizados en una misma máquina pueden compartir una sola memoria caché. El NFS requiere que siempre que un cliente abra un archivo previamente cerrado que haya sido guardado (en parte) en caché, el cliente debe revalidar de inmediato los datos guardados en caché. La revalidación ocurre al verificar cuándo fue modificado por última vez el archivo e invalidando la memoria caché en caso de que contenga datos obsoletos.

En el NFSv4, un servidor puede delegar algunos de sus derechos a un cliente cuando se abre un archivo. La **delegación abierta** ocurre cuando se permite que la máquina cliente maneje localmente las operaciones open y close de otros clientes en la misma máquina. Normalmente, el servidor se encarga de verificar si la apertura de un archivo debe tener éxito o no, por ejemplo, porque las reservaciones de compartimiento tienen que ser tomadas en cuenta. Con la delegación abierta, en ocasiones se permite que la máquina cliente tome dichas decisiones, con ello se evita la necesidad de ponerse en contacto con el servidor.

Por ejemplo, si un servidor ha delegado la apertura de un archivo a un cliente que solicitó permisos de lectura, las solicitudes de bloqueo de un archivo provenientes de otros clientes localizados en la misma máquina también pueden ser manejadas localmente. El servidor seguirá manejando las solicitudes de bloqueo de clientes ubicados en otras máquinas simplemente con negar el acceso al archivo a esos clientes. Observemos que este esquema no funciona en el caso de la delegación de un archivo a un cliente que solicitó sólo permisos de lectura. En ese caso, siempre que otro cliente local desee adquirir permisos de lectura, tendrá que ponerse en contacto con el servidor; no es posible manejar localmente la solicitud.

Una consecuencia importante de delegar un archivo a un cliente es que el servidor tiene que ser capaz de recordar la delegación, por ejemplo, cuando otro cliente situado en una máquina diferente necesita obtener derechos de acceso al archivo. Recordar una delegación requiere que el servidor pueda devolver la llamada al cliente, como se ilustra en la figura 11-22.

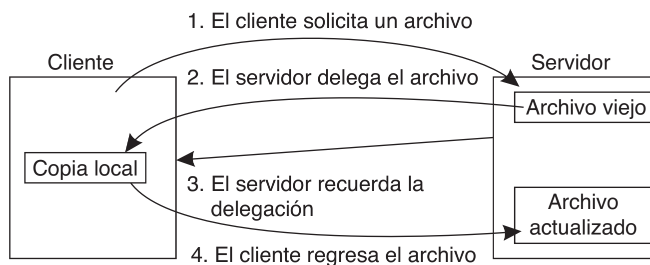


Figura 11-22. Utilización del mecanismo de devolución de llamada en NFSv4 para recordar la delegación de archivos.

En NFS, una llamada de retorno se implementa por medio de sus mecanismos RPC subyacentes. Observe, sin embargo, que las devoluciones de llamada requieren que el servidor no pierda de vista

los clientes a quienes delegó un archivo. Aquí, vemos otro ejemplo donde un servidor NFS ya no puede ser implementado en una forma sin estado. Advierta, sin embargo, que la combinación de delegación y servidores sin estado puede conducir a varios problemas en la presencia de fallas de cliente y servidor. Por ejemplo, ¿qué debe hacer un servidor cuando había delegado un archivo a un cliente ahora irresponsable? Como analizaremos con brevedad, los contratos constituyen generalmente una solución práctica adecuada.

Los clientes también pueden guardar en memoria caché valores de atributo, pero en gran medida se dejan por la paz cuando se trata de conservar consistentes los valores guardados en caché. En particular, los valores de atributo del mismo archivo guardado en caché por dos clientes diferentes pueden ser distintos a menos que los clientes los mantengan mutuamente consistentes. Las modificaciones realizadas a un valor de atributo deben ser remitidas de inmediato al servidor, siguiendo de esa manera una política de coherencia de escritura en la memoria caché.

Se sigue un método similar para guardar en caché manejadores de archivo (o más bien la ruta del manejador del nombre al archivo) y directorios. Para mitigar los efectos de inconsistencias, el NFS utiliza contratos en atributos guardados en caché, manejadores de archivos, y directorios. Después de transcurrido cierto tiempo, las entradas en caché son, por tanto, automáticamente invalidadas y se requiere revalidarlas antes de utilizarlas otra vez.

Almacenamiento en la memoria caché del lado del cliente en Coda

El almacenamiento en la memoria caché del lado del cliente es crucial para la operación de Coda por dos razones. En primer lugar, la finalidad del almacenamiento en caché es lograr escalabilidad. En segundo lugar, el almacenamiento en caché proporciona un alto grado de tolerancia a fallas ya que el cliente se vuelve menos dependiente de la disponibilidad del servidor. Por estas dos razones, en Coda los clientes siempre guardan en caché los archivos completos. En otros términos, cuando se abre un archivo para lectura o escritura, una copia completa se transfiere al cliente, donde posteriormente se guarda en la memoria caché.

A diferencia de muchos otros sistemas de archivo distribuidos, en Coda la coherencia de la memoria caché se mantiene mediante devoluciones de llamadas. Ya enfrentamos este fenómeno cuando estudiamos la semántica de compartimiento de archivos. Por cada archivo, el servidor desde el cual un cliente lo buscó no pierde de vista qué clientes tienen una copia de dicho archivo guardada localmente en caché. Se dice que un servidor registra una **promesa de retorno de llamada** para un cliente. Cuando un cliente actualiza su copia local del archivo por primera vez, lo notifica al servidor, el cual, a su vez, envía un mensaje de invalidación a los demás clientes. Tal mensaje de invalidación se conoce como **ruptura del retorno de llamada**, porque el servidor desechará entonces la promesa de retorno de llamada que mantenía para el cliente al que acaba de enviar una invalidación.

El aspecto interesante de este esquema es que en tanto un cliente sepa que tiene una promesa de devolución de llamada pendiente en el servidor, puede acceder con seguridad al archivo localmente. En particular, supongamos que un cliente abre un archivo y se da cuenta de que aún está en su caché. En ese caso puede utilizarlo siempre que el servidor aún mantenga su promesa de devolución de llamada en relación con el archivo para ese cliente. El cliente deberá comprobar con el servidor si la promesa sigue siendo válida. De ser así, no existe la necesidad de transferir el archivo otra vez del servidor al cliente.

Este método se ilustra en la figura 11-23, la cual es una extensión de la figura 11-20. Cuando un cliente *A* inicia una sesión S_A, el servidor registra una promesa de devolución de llamada. Sucede lo mismo cuando *B* inicia una sesión S_B. Sin embargo, cuando *B* cierra S_B el servidor rompe su promesa de retorno de la llamada al cliente *A* enviando a éste una ruptura de retorno de llamada. Observe que debido a la semántica transaccional de Coda, cuando un cliente *A* cierra la sesión S_A, no sucede nada especial; el cierre simplemente es aceptado, como era de esperarse.

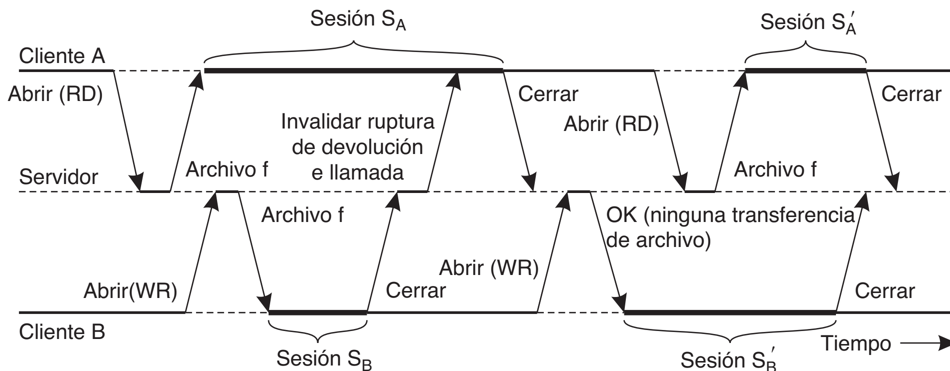


Figura 11-23. Uso de copias locales cuando se abre una sesión en Coda.

La consecuencia es que cuando posteriormente *A* desee abrir la sesión S_A', encontrará que su copia local de *f* fue invalidada, de tal suerte que tendrá que buscar la última versión desde el servidor. Por otra parte, cuando *B* inicie una sesión S_B', notará que el servidor sigue manteniendo una promesa de retorno de llamada pendiente, lo cual implica que *B* simplemente puede reutilizar la copia local que aún conserva de la sesión S_B.

Almacenamiento en la memoria caché del lado del cliente para dispositivos portátiles

Un importante desarrollo para muchos sistemas distribuidos es que ya no se puede asumir que muchos dispositivos de almacenamiento están permanentemente conectados al sistema a través de una red. En lugar de eso, los usuarios tienen varios tipos de dispositivos de almacenamiento semipermanentemente conectados, por ejemplo, mediante bases o estaciones. Ejemplos típicos incluyen PDA, computadoras portátiles y también dispositivos multimedia portátiles tales como reproductores de películas y audio.

En la mayoría de los casos, se utiliza un modelo explícito de carga y descarga para mantener los archivos en dispositivos de almacenamiento portátiles. Las cosas se simplifican si el dispositivo de almacenamiento es visto como parte del sistema de archivo distribuido. En ese caso, siempre que un archivo necesite ser accesado, puede buscarse en el dispositivo local o a lo largo de la conexión hasta el resto del sistema. Se tienen que distinguir estos dos casos.

Tolia y colaboradores (2004) proponen aplicar un método muy simple que consiste en guardar localmente una dispersión (hash) criptográfica de los datos contenidos en archivos. Estas dispersiones se guardan en el dispositivo portátil y se utilizan para redireccionar solicitudes de contenido

asociado. Por ejemplo, cuando se guarda localmente una lista de directorios, en lugar de guardar los datos de cada uno de los archivos listados sólo se guarda el archivo computado. Entonces, cuando un archivo es buscado, el sistema verificará primero si está localmente disponible y actualizado. Observe que un archivo obsoleto tendrá una dispersión diferente del guardado en la lista de directorios. Si el archivo está localmente disponible, puede ser regresado al cliente, de lo contrario se requerirá una transferencia de datos.

Desde luego, cuando un dispositivo se desconecta será imposible transferir cualesquiera datos. Existen varias técnicas para garantizar con una alta probabilidad de éxito que probablemente los archivos a utilizar en realidad sí están guardados localmente en el dispositivo. Comparado con el método de transferencia de datos según la demanda inherente a la mayoría de los esquemas de almacenamiento en memoria caché, en estos casos se tendrían que desplegar técnicas de prebúsqueda de archivos. Sin embargo, para muchos dispositivos de almacenamiento portátiles, es de esperarse que el usuario utilizará programas especiales para preinstalar archivos en el dispositivo.

11.6.2 Replicación del lado del servidor

Por contraste con el almacenamiento en la memoria caché del lado del cliente, en sistemas de archivo distribuidos la replicación del lado del servidor es menos común. Desde luego, la replicación se aplica cuando la disponibilidad está en riesgo, pero desde una perspectiva de desempeño es más sensato desplegar cachés en donde un archivo completo, o partes de él, se ponen localmente disponibles para un cliente. Una importante razón por la cual el almacenamiento en caché del lado del cliente es tan popular es que la práctica muestra que el compartimiento de archivos es relativamente rara. Cuando tiene lugar el compartimiento, a menudo es sólo para leer datos, en cuyo caso el almacenamiento en caché es una excelente solución.

Otro problema con la replicación del lado del servidor en cuanto a desempeño es que una combinación de un alto grado de replicación y una baja proporción de lectura y escritura puede realmente degradar el desempeño. Esto es fácil de entender cuando se cae en la cuenta de que cada operación de actualización tiene que ser realizada en cada réplica. En otros términos, para un archivo replicado de N pliegues, una sola solicitud de actualización conducirá a un incremento de N pliegues de operaciones de actualización. Además, tienen que sincronizarse las operaciones concurrentes, lo cual conduce a más comunicación y más reducción del desempeño.

Por estas razones, los servidores de archivos se replican generalmente para tolerancia a fallas. A continuación, ilustramos este tipo de replicación para el sistema de archivos Coda.

Replicación del servidor en Coda

Coda permite que los servidores se repliquen. Como ya mencionamos, la unidad de replicación es un conjunto de archivos llamado **volumen**. En esencia, un volumen corresponde a una partición de disco UNIX, es decir, un sistema de archivo tradicional como los soportados directamente por los sistemas operativos, aunque los volúmenes generalmente son mucho más pequeños. El conjunto de servidores Coda que guardan una copia de un volumen se conoce como **grupo de almacenamiento de volumen**, o simplemente **VSG** (por sus siglas en inglés). En la presencia de fallas, es posible que un cliente no pueda acceder a todos los servidores incluidos en un VSG de volumen. Un **grupo**

de almacenamiento de volumen accesible (**AVSG**, por sus siglas en inglés) de cliente consta de aquellos servidores presentes en el VSG de volumen que el cliente puede contactar al momento. Si el AVSG está vacío, se dice que el cliente está **desconectado**.

Coda utiliza un protocolo de escritura replicado para mantener la consistencia de un volumen replicado. En particular, utiliza una variante de Read-One, Write-All (ROWA), el cual se explicó en el capítulo 7. Cuando un cliente necesita leer un archivo, se pone en contacto con uno de los miembros de sus AVSG del volumen al cual pertenece el archivo. Sin embargo, cuando en un archivo actualizado se cierra una sesión, el cliente lo transfiere en paralelo a cada miembro del AVSG. Esta transferencia en paralelo se logra por medio de MultiRPC como explicamos antes.

Este esquema funciona bien en tanto no ocurran fallas, es decir, para cada cliente, su AVSG de un volumen es el mismo que su VSG. Sin embargo, en presencia de fallas, las cosas pueden ir mal. Consideremos un volumen replicado a través de tres servidores S_1, S_2 y S_3. Para el cliente A, suponga que su AVSG comprende los servidores S_1 y S_2 en tanto que el cliente B tiene acceso únicamente al servidor S_3, como se muestra en la figura 11-24.

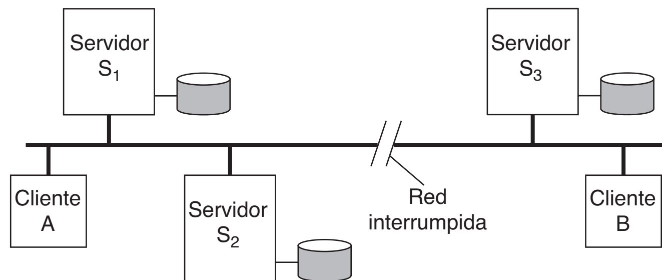


Figura 11-24. Dos clientes con un AVSG diferente para el mismo archivo replicado.

Coda utiliza una estrategia optimista para la replicación de archivos. En particular, tanto A como B tendrán permiso de abrir un archivo, f, para escribir, actualizar sus copias respectivas, y transferir su copia de vuelta a los miembros en sus AVSG. Desde luego, habrá diferentes versiones de f guardadas en el VSG. La pregunta es cómo se puede detectar y resolver esta inconsistencia.

La solución adoptada por Coda es desplegar un esquema generador de versiones. En particular, en un VSG un servidor S_i mantiene un **vector de versión Coda** $CVV_i(f)$ por cada archivo f contenido en el VSG. Si $CVV_i(f)[j] = k$, entonces el servidor S_i sabe que el servidor S_j ha visto por lo menos la versión k del archivo f. $CVV_i(f)[i]$ es el número de la versión actual de f guardada en el servidor S_i. Una actualización de f en el servidor S_i conducirá a un incremento de $CVV_i(f)[i]$. Observe que los vectores de versión son completamente análogos a los registros de tiempo vectoriales presentados en el capítulo 6.

Volvamos al ejemplo de tres servidores. $CVV_i(f)$ es inicialmente igual a [1, 1, 1] para cada servidor S_i. Cuando el cliente A lee f desde uno de los servidores en su AVSG, por ejemplo S_1, también recibe $CVV_1(f)$. Después de actualizar f, el cliente A multitransmite f hacia cada servidor en

su AVSG, es decir, S_1 y S_2. Ambos servidores registrarán entonces que su respectiva copia ha sido actualizada, pero no la de S_3. En otros términos,

$$CVV_1(f) = CVV_2(f) = [2, 2, 1]$$

Mientras tanto, se permitirá que el cliente *B* abra una sesión en la cual recibe una copia de *f* del servidor S_3, y posteriormente actualiza también a *f*. Cuando cierre su sesión y transfiera la actualización a S_3, el servidor S_3 actualizará su vector de versión a $CVV_3(f) = [1, 1, 2]$.

Cuando la partición cicatrice, los tres servidores tendrán que reintegrar sus copias de *f*. Al comparar sus vectores de versión, advertirán que se ha presentado un conflicto que debe ser reparado. En muchos casos, la resolución de conflictos puede ser automática en una forma dependiente de una aplicación, como analizan Kumar y Satyanarayanan (1995). Sin embargo, también existen muchos casos en que los usuarios tendrán que ayudar a resolver un conflicto manualmente, en especial cuando diferentes usuarios han cambiado la misma parte del mismo archivo de diferentes maneras.

11.6.3 Replicación en sistemas de archivo punto a punto

A continuación examinaremos la replicación en sistemas de archivos compartidos punto a punto. Aquí, la replicación también desempeña un rol importante, principalmente para acelerar las solicitudes de búsqueda, pero también para equilibrar la carga entre nodos. Una propiedad importante incluida en estos sistemas es que virtualmente todos los archivos son de sólo lectura. Las actualizaciones consisten únicamente en la forma de agregar archivos al sistema. Se deberá hacer una distinción entre sistemas punto a punto estructurados y no estructurados.

Sistemas punto a punto no estructurados

Fundamental para los sistemas punto a punto no estructurados es que la búsqueda de datos se reduce a *buscarlos* en la red. Efectivamente, esto significa que un nodo tendrá simplemente, por ejemplo, que transmitir una solicitud de búsqueda a sus vecinos, desde donde la búsqueda pueda ser remitida y así sucesivamente. Desde luego, buscar mediante transmisión generalmente no es una buena idea y se tienen que tomar medidas especiales para evitar problemas de desempeño. La búsqueda en sistemas punto a punto se analiza extensamente en Risson y Moors (2006).

Independiente de la forma la transmisión está limitada, deberá quedar claro que si se replican los archivos la búsqueda se vuelve más fácil y rápida. Un extremo es para replicar un archivo en todos los nodos, lo cual implicaría que la búsqueda de cualquier archivo pueda ser realizada enteramente de modo local. Sin embargo, dado que los nodos tienen capacidad limitada, la replicación completa es imposible. El problema es entonces encontrar una estrategia de replicación óptima, donde la excelencia está definida por el número de los diferentes nodos que deben procesar una búsqueda específica antes de encontrar un archivo.

Cohen y Shenker (2002) han examinado este problema, suponiendo que la replicación de archivos puede ser controlada. En otros términos, suponiendo que en un sistema punto a punto no

estructurado los nodos pueden ser instruidos para que mantengan copias de archivos, ¿cuál es entonces la mejor forma de asignación de copias de un archivo a los nodos?

Consideremos dos extremos. Una política es distribuir uniformemente *n* copias de cada archivo a través de toda la red. Esta política ignora que diferentes archivos pueden tener diferentes tasas de solicitud, es decir, algunos archivos son más populares que otros. Como alternativa, otra política es replicar archivos de acuerdo con qué tan a menudo son buscados: mientras más popular es un archivo, más réplicas se crean y distribuyen a través de la sobrecapa.

Como un comentario al margen, observemos que esta última política puede hacer que resulte muy caro localizar archivos no populares. Por extraño que esto parezca, tales búsquedas pueden demostrar ser cada vez más importantes desde un punto de vista económico. El razonamiento es simple: con una internet que permite un rápido y fácil acceso a toneladas de información, la explotación de nichos de mercado de repente se vuelve atractiva. Así que, si usted desea obtener el equipo apropiado para, por ejemplo, una bicicleta reclinada, internet es el lugar idóneo para localizarlo ya que sus medios de búsqueda le permitirán descubrir con eficiencia al vendedor adecuado.

Bastante sorprendente resulta que la política uniforme y la política popular sean igual de buenas cuando buscan el número promedio de nodos a ser solicitados. La distribución de las búsquedas es igual en ambos casos, y de tal modo que la distribución de documentos en la política popular sigue a la distribución de búsquedas. Además, resulta que cualquier asignación “entre” estas dos políticas es mejor. Obtener tal asignación es factible, pero no trivial.

La replicación en sistemas punto a punto no estructurados sucede naturalmente cuando los usuarios descargan archivos a partir de otros usuarios y luego los ponen a disposición de la comunidad. El control de estas redes es muy difícil en la práctica, excepto cuando algunas partes son controladas por una sola organización. Además, como lo indican estudios realizados sobre BitTorrent, también existe un importante factor social cuando se trata de replicar archivos y ponerlos a disposición (Pouwelse y cols., 2005). Por ejemplo, algunas personas muestran una conducta altruista, o simplemente continúan haciendo que los archivos estén disponibles por no más tiempo que el necesario una vez que han completado su descarga. La pregunta que viene a la mente es si se pueden idear sistemas que exploten este comportamiento.

Sistemas estructurados punto a punto

Al considerar la eficiencia de las operaciones de búsqueda en sistemas punto a punto estructurados, la replicación es desplegada principalmente para equilibrar la carga entre los nodos. En el capítulo 5 vimos cómo una forma “estructurada” de replicación, tal como fue explotada por Ramasubramanian y Sirer (2004b) podría incluso reducir los pasos de búsqueda promedio a O(1). Sin embargo, cuando se trata de balancear la carga, se tienen que explorar métodos diferentes.

Un método aplicado a menudo es simplemente replicar la ruta que una búsqueda siguió desde su origen hasta su destino. Esta política de replicación tendrá el efecto de que la mayoría de las réplicas se colocarán cerca del nodo responsable de guardar el archivo, y entonces descargarán dicho nodo cuando haya una alta tasa de solicitudes. Sin embargo, tal política de replicación no toma en cuenta la carga de otros nodos, y por tanto puede conducir a un sistema desbalanceado.

Para abordar estos problemas, Gopalakrishnan y colaboradores (2004) proponen un esquema diferente que toma en cuenta la carga actual de los nodos a lo largo de la ruta de búsqueda. La idea principal es guardar réplicas en el nodo origen de una búsqueda, y guardar los apuntadores a tales réplicas en la memoria caché en nodos localizados a lo largo de la ruta de búsqueda desde el origen hasta el destino. Más específicamente, cuando la ruta de búsqueda del nodo *P* al *Q* pasa a través del nodo *R*, éste verificará si cualquiera de *sus* archivos debe ser descargado a *P*. Esto lo hace buscando simplemente en su propia carga de búsqueda. Si *R* se ocupa de demasiadas solicitudes de búsqueda de archivos que actualmente está guardando en comparación con la carga impuesta en *P*, se puede pedir que *P* instale copias de los archivos más solicitados de *R*. Este principio se ilustra en la figura 11-25.

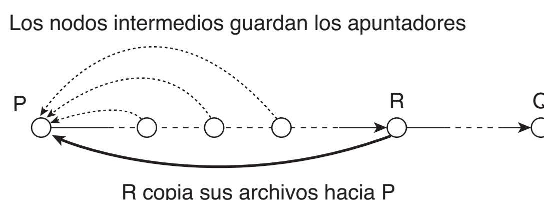


Figura 11-25. Balanceo de carga en un sistema punto a punto mediante replicación.

Si *P* puede aceptar el archivo *f* de *R*, cada nodo visitado en la ruta de *P* a *R* instalará un apuntador para *f* a *P*, para indicar que en *P* se encuentra una réplica de *f*.

Queda claro que la diseminación de información sobre dónde están guardadas las réplicas es importante para que este esquema funcione. Por consiguiente, cuando se enruta una búsqueda a través de la sobrecapa, un nodo también puede transferir información con respecto a la réplicas que está alojando. Esta información puede conducir entonces a una instalación adicional de apuntadores, lo cual permite que los nodos tomen decisiones informadas de redireccionamiento de solicitudes hacia nodos que mantienen una réplica de un archivo solicitado. Estos apuntadores se colocan en un caché de tamaño limitado y son reemplazados siguiendo una política simple del menos recientemente utilizado (es decir, los apuntadores guardados en caché que se refieran a archivos nunca solicitados, serán eliminados de inmediato).

11.6.4 Replicación de archivos en sistemas de Malla (Grid)

Como último tema relacionado con la replicación de archivos, consideremos lo que sucede en la computación Grid. Naturalmente, el desempeño juega un rol tan crucial en esta área que muchas aplicaciones Grid son altamente intensivas en cuanto a cómputo. Además, vemos que las aplicaciones con frecuencia también necesitan procesar vastas cantidades de datos. Por consiguiente, se ha puesto mucho esfuerzo en la replicación de archivos en los cuales se están ejecutando aplicaciones. Los medios para hacerlo, sin embargo, son sorprendentemente simples.

Una observación fundamental es que muchos datos de aplicaciones Grid son de sólo lectura. Los datos a menudo son producidos por sensores, o con otras aplicaciones, pero rara vez son actua-

lizados o modificados después de que son producidos y guardados. Por consiguiente, la replicación de datos puede ser aplicada en abundancia, y esto es exactamente lo que sucede.

Por desgracia, el tamaño de los conjuntos de datos en ocasiones es tan enorme que se deben tomar medidas especiales para evitar que los proveedores de datos (es decir, las máquinas que guardan los conjuntos de datos) se sobrecarguen debido a la cantidad de datos que tienen que transferir a través de la red. Por otra parte, como muchos de los datos están excesivamente replicados, el balanceo de la carga para recuperar copias es un asunto menor.

En los sistemas de malla (Grid), la replicación evolucionó principalmente en torno al problema de localizar las mejores fuentes para copiar datos. Este problema se resuelve mediante **servicio de localización de réplicas**, muy similar a los servicios de localización presentados en el caso de sistemas de asignación de nombre. Un método evidente que ha sido desarrollado para el juego de herramientas Globus es utilizar un sistema basado en DHT tal como el sistema de cuerdas para la búsqueda descentralizada de réplicas (Cai y cols., 2004). En este caso, un cliente transfiere un nombre de archivo a cualquier nodo del servicio, donde es transformado en una clave y posteriormente examinado. La información regresada al cliente contiene direcciones de contacto de los archivos solicitados.

Para mantener las cosas simples, los archivos localizados se descargan posteriormente desde varios sitios mediante un protocolo FTP, después de lo cual el cliente puede registrar sus propias réplicas con el servicio de localización de réplicas. Esta arquitectura se describe con más detalle en Chervenak y colaboradores (2005), pero el método es bastante simple.

11.7 TOLERANCIA A FALLAS

En los sistemas de archivo distribuidos, la tolerancia a fallas se maneja de acuerdo con el principio que analizamos en el capítulo 8. Como ya se mencionó, en muchos casos, la replicación se realiza para crear grupos de servidores tolerantes a fallas. En esta sección, por consiguiente, se pondrá énfasis en algunos temas especiales de tolerancia a fallas en sistemas de archivo distribuidos.

11.7.1 Manejo de fallas bizantinas

Uno de los problemas que a menudo se pasa por alto cuando se trata de tolerancia a fallas es que los servidores pueden exhibir fallas arbitrarias. En otros términos, la mayoría de los sistemas no considera las fallas bizantinas estudiadas en el capítulo 8. Además de su complejidad, la razón para ignorar este tipo de fallas tiene que ver con las fuertes suposiciones que se deben hacer con respecto al entorno de ejecución. De modo notable, se debe suponer que las demoras en la comunicación están delimitadas.

En entornos prácticos, tal suposición no es realista. Por esta razón, Castro y Liskov (2002) idearon una solución para manejar fallas bizantinas que también puede operar en redes tales como internet. Aquí analizamos este protocolo, ya que puede ser (y ha sido) aplicado directamente a sistemas de archivo distribuidos, en forma notable a un sistema basado en NFS. Desde luego, también existen otras aplicaciones. La idea básica es aplicar la replicación activa construyendo un conjunto de máquinas de estado finito y hacer que los procesos no defectuosos presentes en este conjunto ejecuten operaciones en el mismo orden. Suponiendo que cuando mucho k procesos fallan a la vez, un

cliente envía una operación a todo el grupo y acepta la respuesta de por lo menos $k + 1$ procesos diferentes.

Para lograr protección contra fallas bizantinas, el grupo de servidores debe componerse de por lo menos $3k + 1$ procesos. La parte difícil para lograr esta protección es asegurarse de que todos los procesos no defectuosos ejecuten todas las operaciones en el mismo orden. Una forma simple de alcanzar este objetivo es asignar un coordinador que simplemente ponga en serie todas las operaciones anexando un número de secuencia a cada solicitud. El problema, desde luego, es que el coordinador puede fallar.

Los problemas inician con los coordinadores que fallan. Muchísimo sucede como con la sincronía virtual, es decir, los procesos atraviesan por una serie de visiones donde en cada visión los miembros se ponen de acuerdo en los procesos no defectuosos e inician un cambio de visión cuando el maestro actual parece estar fallando. Esto último puede ser detectado si se asume que los números en secuencia son entregados uno después del otro, de modo que una brecha, o un tiempo fuera en el caso de una operación, puede indicar que algo anda mal. Observemos que los procesos pueden concluir falsamente que debe instalarse una nueva visión. Sin embargo, esto no afectará la corrección del sistema.

Una parte importante del protocolo depende de que las solicitudes pueden ordenarse correctamente. Con este objeto, se utiliza un mecanismo de quórum: siempre que un proceso recibe una solicitud de ejecutar una operación o con el número n en la visión v, envía esta información a todos los procesos y espera hasta que recibe la confirmación de por lo menos $2k$ procesos que recibieron la misma solicitud. De este modo, se obtiene un quórum de tamaño $2k + 1$ para la solicitud. Tal confirmación se llama **certificado de quórum**. En esencia, dice que un número suficientemente grande de procesos han guardado la misma solicitud, y que por tanto es seguro proseguir.

El protocolo completo consta de cinco fases, y se muestra en la figura 11-26.

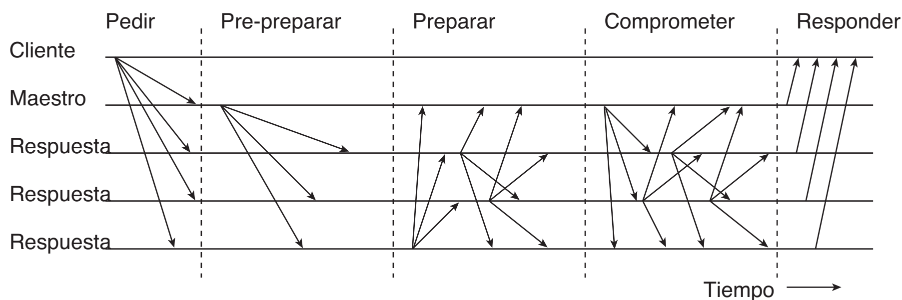


Figura 11-26. Diferentes fases en la tolerancia a fallas bizantinas.

Durante la primera fase, un cliente envía una solicitud a todo el grupo de servidores. Una vez que el maestro ha recibido la solicitud, multitransmite un número en secuencia en la *fase de pre-preparar*, de modo que la operación asociada será apropiadamente ordenada. En ese punto, las réplicas esclavas tienen que garantizar que el número en secuencia del maestro sea aceptado por un quórum, siempre que cada réplica acepte la propuesta del maestro. Por consiguiente, si una esclava

acepta el número en secuencia propuesto, multitransmite esta aceptación a las demás. Durante la fase de *realización*, se llega a un acuerdo y todos los procesos se pasan la información entre sí y ejecutan la operación, tras de lo cual el cliente finalmente puede ver el resultado.

Cuando se consideran las diversas fases, puede parecer que después de la fase de *preparación* todos los procesos deberán estar de acuerdo en el mismo orden de las solicitudes. Sin embargo, esto es cierto sólo dentro de la misma visión: si hubiera la necesidad de cambiar a una nueva visión, diferentes procesos pueden tener el mismo número de secuencia para diferentes operaciones, las cuales fueron asignadas en diferentes visiones. Por esta razón, también se tiene la fase de *realización*, en la cual cada proceso informa entonces a los demás procesos que ha guardado la solicitud en su registro local y para la visión actual. Por consiguiente, incluso si no existe la necesidad de recuperarse de una congelación, un proceso sabrá con exactitud qué número de secuencia le había sido asignado y durante qué visión.

De nueva cuenta, una operación comprometida puede ser ejecutada en cuanto un proceso no defectuoso ha visto los mismos $2k$ mensajes (y éstos deberán concordar con sus propias intenciones). De nueva cuenta, ahora se tiene un quórum de $2k + 1$ para ejecutar la operación. Desde luego, las operaciones pendientes con números de secuencia más bajos deberán ser ejecutadas primero.

El cambio a una nueva visión sigue en esencia los cambios de visión para sincronía virtual descritos en el capítulo 8. En este caso, un proceso tiene que enviar información sobre los mensajes pre-preparados de los que tiene conocimiento, así como también de los mensajes preparados recibidos de la visión previa. Aquí pasaremos por alto los detalles.

El protocolo ha sido implementado para un sistema de archivos basado en NFS, junto con varias optimizaciones importantes y estructuras de datos cuidadosamente diseñadas, cuyos detalles se encuentran en Castro y Liskov (2002). Una descripción de un envolvedor que permite incorporar la tolerancia a fallas bizantinas con aplicaciones heredadas se localiza en Castro y colaboradores (2003).

11.7.2 Alta disponibilidad en sistemas punto a punto

Un tema que ha recibido especial atención es garantizar la disponibilidad en sistemas punto a punto. Por un lado, parecería que simplemente con la replicación de archivos la disponibilidad resulta fácil de garantizar. El problema, sin embargo, es que la no disponibilidad de nodos es tan alta que este razonamiento simple ya no es válido. Como explicamos en el capítulo 8, la solución clave a la alta disponibilidad es la redundancia. Cuando se trata de archivos, básicamente hay dos métodos diferentes de obtener redundancia: replicación y codificación de borradura.

La **codificación de borradura** es una técnica muy conocida mediante la cual un archivo se divide en dos m fragmentos que posteriormente se recodifican en $n > m$ fragmentos. El problema crucial en este esquema de codificación es que cualquier conjunto de m fragmentos codificados es suficiente para reconstruir el archivo original. En este caso, el factor de redundancia es igual a $r_{ec} = n/m$. Suponiendo una disponibilidad de nodos promedio de a, y una no disponibilidad de archivos requerida de ε, se debe garantizar que por lo menos m fragmentos estén disponibles, es decir:

$$1 - \varepsilon = \sum_{i=m}^{n} \begin{bmatrix} n \\ i \end{bmatrix} a^i (1-a)^{n-i}$$

Si comparamos esto con la replicación de archivos, vemos que la no disponibilidad de archivos es completamente dictada por la probabilidad de que todas sus réplicas r_{rep} estén no disponibles. Si asumimos que las partidas de nodos son independientes y están idénticamente distribuidas, tenemos

$$1 - \varepsilon = 1 - (1 - a)^{r_{rep}}$$

Al aplicar algunas manipulaciones y aproximaciones algebraicas, podemos expresar la diferencia entre replicación y codificación de borradura considerando la relación r_{rep}/r_{ec} en su relación con la disponibilidad de a de los nodos. Esta relación se muestra en la figura 11-27, donde hicimos $m = 5$ [vea también Bhagwan y colaboradores (2004), y Rodrigues y Liskov (2005)].

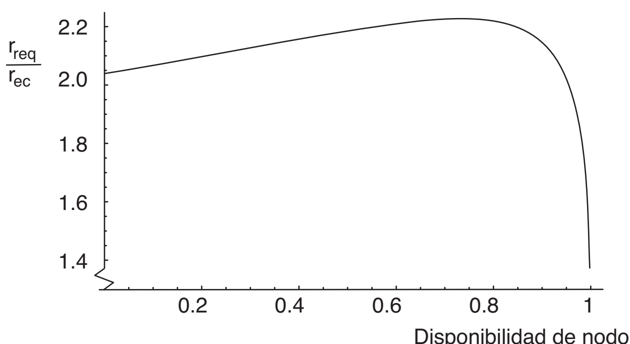


Figura 11-27. Relación r_{rep}/r_{ec} como una función de la disponibilidad del nodo a.

En esta figura observamos que bajo todas las circunstancias, la codificación de borradura requiere menos redundancia que la replicación de archivos. En otros términos, la replicación de archivos para incrementar la disponibilidad en redes punto a punto, donde los nodos regularmente llegan y se van, es menos eficiente desde una perspectiva de almacenamiento que utiliza técnicas de codificación de borradura.

Se puede argumentar que estos ahorros en espacio de almacenamiento en realidad ya no son un problema puesto que la capacidad en disco a menudo resulta abrumadora. Sin embargo, cuando se cae en la cuenta de que mantener la redundancia impondrá comunicación, entonces una redundancia más baja ahorra ancho de banda. Esta ganancia de desempeño es extraordinariamente importante cuando los nodos corresponden a máquinas de usuario conectadas a internet mediante líneas de cables o DSL, donde los vínculos de salida a menudo tienen capacidad de sólo algunos cientos de Kbps.

11.8 SEGURIDAD

Muchos de los principios de seguridad analizados en el capítulo 9 se aplican directamente a sistemas de archivo distribuidos. La seguridad en sistemas de archivo distribuidos organizada a lo largo de una arquitectura cliente-servidor es hacer que los servidores manejen la autenticación y el control

de acceso. Ésta es una forma simple de ocuparse de la seguridad, un método que ha sido adoptado, por ejemplo, en sistemas tales como NFS.

En esos casos, es común contar con un servicio de autenticación aparte, tal como Kerberos, en tanto que los servidores de archivo simplemente se encargan de la autorización. Una desventaja importante de este esquema es que requiere administración centralizada de los usuarios, lo cual puede afectar severamente la escalabilidad. A continuación, analizamos brevemente la seguridad en NFS como un ejemplo del método tradicional, después de lo cual prestamos atención a métodos alternativos.

11.8.1 Seguridad en NFS

Como ya mencionamos, la idea básica detrás del NFS es que un sistema de archivo remoto deberá ser presentado a los clientes como si fuera un sistema de archivo local. Desde este punto de vista, no sorprende que la seguridad en NFS se enfoque principalmente en la comunicación entre un cliente y un servidor. Comunicación segura significa que se deberá establecer un canal seguro entre los dos, tal como vimos en el capítulo 9.

Además de RPC seguras, es necesario controlar los accesos a archivos, que en NFS son manejados por atributos de archivo de control de acceso. Un servidor de archivos se encarga de verificar los derechos de acceso de sus clientes, como se explicará a continuación. Combinada con RPC seguras, la arquitectura de seguridad NFS se muestra en la figura 11-28.

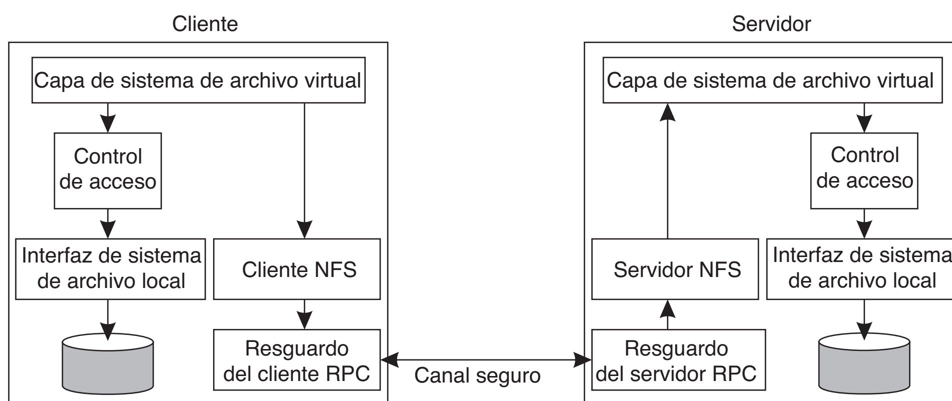


Figura 11-28. Arquitectura de seguridad NFS.

RPC seguras

Como el NFS está colocado sobre un sistema RPC, el establecimiento de un canal seguro en NFS se reduce a establecer RPC seguras. Hasta el NFSv4, una RPC segura significaba que sólo se ocupaba de la autenticación. Había tres formas de llevar a cabo la autenticación. Enseguida examinaremos cada una de dichas formas.

El método más utilizado, uno que en la actualidad rara vez se usa, se conoce como autenticación de sistema. En este método basado en UNIX, un cliente simplemente transfiere sus ID de usuario y de grupo efectivas al servidor, junto con una lista de grupos de los que afirma ser miembro. Esta información se envía al servidor como texto común sin firmar. En otros términos, el servidor no tiene forma en absoluto de verificar si los identificadores de usuario y grupo están realmente asociados con el remitente. En esencia, el servidor asume que el cliente transfirió un procedimiento de inicio de sesión apropiado, y que puede confiar en la máquina del cliente.

El segundo método de autenticación en versiones de NFS más viejas utiliza un intercambio de clave Diffie-Hellman para establecer una clave de sesión, ello conduce a lo que se llama **NFS seguro**. En el capítulo 9 explicamos cómo funciona el intercambio de clave Diffie-Hellman. Este método es mejor que el de autenticación de sistema, aunque más complejo, razón por la cual se implementa con menos frecuencia. El Diffie-Hellman puede ser visto como un criptosistema de clave pública. Inicialmente, no había forma de distribuir con seguridad la clave pública de un servidor, pero esto se corrigió posteriormente con la introducción de un servicio de nombres seguro. El uso de claves públicas relativamente pequeñas siempre ha sido un punto de crítica, estas claves sólo son de 192 bits en NFS. Se ha demostrado que la irrupción de un sistema Diffie-Hellman con esas claves tan cortas es casi trivial (Lamacchia y Odlyzko, 1991).

El tercer protocolo de autenticación es Kerberos, el cual también describimos en el capítulo 9.

Con la introducción del NFSv4, la seguridad mejoró por medio del soporte de RPCSEC_GSS. El **RPCSEC_GSS** es una estructura general de seguridad que puede soportar una miríada de mecanismos de seguridad para el establecimiento de canales seguros (Eisler y cols., 1997). En particular, no sólo proporciona los medios para implementar diferentes sistemas de autenticación, también soporta integridad y confidencialidad de los mensajes, dos características que no eran soportadas por versiones más viejas de NFS.

El RPCSEC_GSS está basado en una interfaz estándar para servicios de seguridad, denominada **GSS-API**, la cual se describe a cabalidad en Linn (1997). El RPCSEC_GSS está colocado encima de esta interfaz, lo cual conduce a la organización mostrada en la figura 11-29.

Para el NFSv4, el RPCSEC_GSS deberá configurarse con soporte para Kerberos V5. Además, el sistema también debe soportar un método conocido como **LIPKEY**, descrito en Eisler (2000). El LIPKEY es un sistema de clave pública que permite autenticar los clientes con una contraseña, en tanto que los servidores son autenticados mediante una clave pública.

En NFS, el aspecto importante de una RPC segura es que los diseñadores hayan decidido no proporcionar sus propios mecanismos de seguridad, sino que sólo proporcionen una forma estándar de manejar la seguridad. Por consiguiente, mecanismos de seguridad comprobados, tales como Kerberos, pueden ser incorporados a una implementación NFS sin afectar otras partes del sistema. También, si resulta que los mecanismos de seguridad existentes tienen fallas (como en el caso de Diffie-Hellman cuando se utilizan claves pequeñas), pueden ser fácilmente reemplazados.

Es importante destacar que, como el RPCSEC_GSS se implementa como parte de la capa RPC que sirve de fundamento para los protocolos NFS, también puede ser utilizado para versiones más

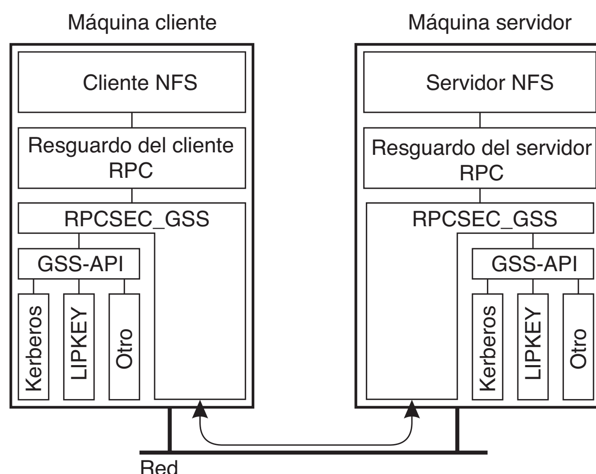


Figura 11-29. RPC segura en NFSv4.

antiguas de NFS. Sin embargo, esta adaptación de la capa RPC estuvo disponible sólo con la introducción del NFSv4.

Control de acceso

En NFS la autorización es análoga a la RPC segura: proporciona los mecanismos pero no especifica ninguna política en particular. El control de acceso es soportado por el atributo de archivo *ACL*. Este atributo es una lista de entradas de control de acceso, donde cada entrada especifica los derechos de acceso para un usuario o grupo específico. Muchas de las operaciones que NFS distingue con respecto a control de acceso son relativamente simples e incluyen las de lectura, escritura, ejecución de archivos, manipulación de atributos de archivo, listados de directorios y así sucesivamente.

La operación synchronize también es digna de atención, la cual en esencia establece si un proceso que está colocado en un servidor puede acceder directamente a un archivo, evitando el protocolo NFS para un mejor desempeño. El modelo NFS de control de acceso tiene una semántica mucho más rica que la mayoría de los modelos UNIX. Esta diferencia se deriva de los requerimientos de que el NFS debe ser capaz de interoperar con sistemas Windows. La idea subyacente es que resulta mucho más fácil adaptarse al modelo UNIX de control de acceso que al de Windows.

Otro aspecto que distingue el control de acceso de sistemas de archivo tales como UNIX, es que el acceso puede ser especificado para un solo usuario (el propietario del archivo), un solo grupo de usuarios (por ejemplo, miembros de un equipo de proyecto) y para todo mundo. El NFS tiene muchas clases diferentes de usuarios y procesos, como se muestra en la figura 11-30.

Tipo de usuario	Descripción
Propietario	El propietario de un archivo
Grupo	El grupo de usuarios asociado con un archivo
Todos	Cualquier usuario o proceso
Interactivo	Cualquier proceso que accede al archivo desde una terminal interactiva
Red	Cualquier proceso que accede al archivo vía la red
Marcado	Cualquier proceso que accede al archivo a través de una conexión telefónica al servidor
Lotes	Cualquier proceso que accede al archivo como parte de un trabajo por lotes
Anónimo	Cualquiera que accede al archivo sin autenticación
Autenticado	Cualquier usuario o proceso autenticado
Servicio	Cualquier proceso de servicio definido por el sistema

Figura 11-30. Diversas clases de usuarios y procesos distinguidos por NFS con respecto al control de acceso.

11.8.2 Autenticación descentralizada

Uno de los problemas principales con sistemas como el NFS es que para manejar apropiadamente la autenticación, es necesario que los usuarios se registren mediante una administración de sistema central. Una solución a este problema es provista por los **sistemas de archivo seguro** (**SFS**, por sus siglas en inglés) en combinación con servidores de autenticación descentralizados. La idea básica descrita con todo detalle en Kaminsky y colaboradores (2003) es bastante simple. De lo que carecen otros sistemas es de la probabilidad de que un usuario especifique que un usuario *remoto* tiene ciertos privilegios sobre sus archivos. En virtualmente todos los casos, los usuarios deben ser globalmente conocidos por todos los servidores de autenticación. Un método más simple sería permitir que Alicia especifique que “Bob cuyos detalles se encuentran en *X*”, tiene ciertos privilegios. El servidor de autenticación que maneja las credenciales de Alicia podría entonces ponerse en contacto con el servidor *S* para obtener información sobre Bob.

Un problema importante a resolver es hacer que el servidor de Alicia sepa con seguridad que se trata del servidor de autenticación de Bob. Este problema se resuelve con nombres de autocertificantes, un concepto introducido en SFS (Mazières y cols., 1999) que pretende separar la gestión de clave de la seguridad de un sistema de archivo. La organización total del SFS aparece en la figura 11-31. Para garantizar la portabilidad a través de un amplio rango de máquinas, varios componentes NFSv3 se integraron al SFS. En la máquina del cliente, existen tres componentes diferentes, sin contar el programa de usuario. El cliente NFS se utiliza como interfaz para comunicarse con los programas de usuario e intercambiar información con un **cliente SFS**. Éste aparece ante el cliente NFS como si fuera otro servidor NFS.

El cliente SFS es responsable de establecer canales seguros con un servidor SFS. También es responsable de comunicarse con un **agente de usuario SFS**, el cual es un programa que maneja automáticamente la autenticación de un usuario. El SFS no prescribe cómo deberá ocurrir la auten-

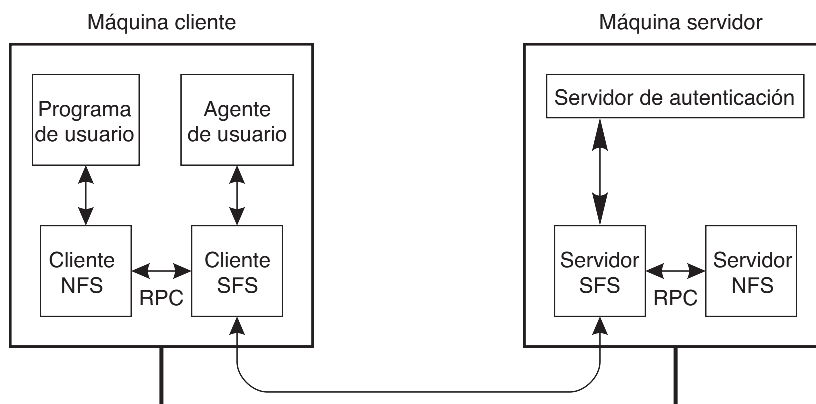


Figura 11-31. Organización del SFS.

ticación del usuario. En correspondencia con sus objetivos de diseño, el SFS separa cuestiones como esas y utiliza distintos agentes para diferentes protocolos de autenticación de usuarios.

Del lado del servidor también existen tres componentes. El servidor NFS se utiliza otra vez por razones de portabilidad. Este servidor se comunica con el **servidor SFS** que opera como un cliente NFS en relación con el servidor NFS. El servidor SFS conforma el proceso central del SFS. Este proceso es responsable de manejar solicitudes de archivo de clientes SFS. Análogo al agente SFS, un servidor SFS se comunica con un servidor de autenticación aparte para que maneje la autenticación de usuarios.

Lo que hace único al SFS en comparación con otros sistemas de archivo distribuidos es la organización de su espacio para nombres. El SFS proporciona un espacio global para nombres enraizado en un directorio llamado */sfs*. Un cliente SFS permite a sus usuarios crear vínculos simbólicos dentro de su espacio de nombre. Aún más importante, el SFS utiliza **nombres de ruta autocertificantes** para nombrar sus archivos. El nombre de ruta porta toda la información necesaria para autenticar el servidor SFS que proporciona el archivo nombrado. Un nombre de ruta autocertificante consta de tres partes, como se muestra en la figura 11-32.



Figura 11-32. Nombre de ruta autocertificante en SFS.

La primera parte del nombre se compone de una ubicación *LOC*, la cual es o un nombre de dominio DNS que identifica el servidor SFS, o su dirección IP correspondiente. El SFS asume que cada servidor *S* tiene una clave pública K_S^+. La segunda parte de un nombre de ruta autocertificante

es un identificador de servidor *HID* que se calcula tomando una dispersión (hash) criptográfica *H* que incluya la ubicación del servidor y su clave pública:

$$HID = H(LOC, K_S^+)$$

HID está representado por un número de 32 dígitos en base 32. La tercera parte está formada por el nombre de ruta local implementado en el servidor SFS y bajo el cual el archivo está realmente guardado.

Siempre que un cliente accede a un servidor SFS, puede autenticar al servidor preguntándole simplemente por su clave pública. Con la bien conocida función hash *H*, el cliente puede calcular entonces el *HID* y verificarlo contra el valor encontrado en el nombre de ruta. Si los dos elementos coinciden, el cliente sabe que está hablando con el servidor que ostenta el nombre encontrado en la ubicación.

¿Cómo separa este método la gestión de clave de la seguridad del sistema de archivo? El problema que el SFS resuelve es que la obtención de una clave pública puede apartarse por completo de los temas de seguridad de un sistema de archivo. Una forma de obtener la clave del servidor es permitiendo que un cliente se ponga en contacto con un servidor y solicite la clave, como ya se describió. Sin embargo, también es posible guardar localmente un conjunto de claves, por ejemplo, mediante los administradores del sistema. En este caso, no se tiene que contactar un servidor. En cambio, cuando se resuelve un nombre de ruta, se busca localmente la clave del servidor, tras de lo cual el ID del servidor puede ser verificado usando la parte que corresponde a la ubicación del nombre de ruta.

Para simplificar las cosas, en la asignación de nombres la transparencia se logra con el uso de vínculos simbólicos. Por ejemplo, supongamos que un cliente desea acceder a un archivo llamado

/sfs/sfs.cs.vu.nl:ag62hty4wior450hdh63u62i4f0kqere/home/steen/mbox

Para ocultar el ID del servidor, un usuario puede crear un vínculo simbólico

/sfs/vucs → */sfs/sfs.cs.vu.nl:ag62hty4wior450hdh63u62i4f0kqere*

y posteriormente utilizar sólo el nombre de ruta */sfs/vucs/home/steen/mbox*. La resolución de dicho nombre se expandirá automáticamente a todo el nombre de ruta SFS, y utilizando la clave pública localmente encontrada, autenticará el servidor SFS de nombre *sfs.vu.cs.nl*.

De manera similar, el SFS puede ser soportado por autoridades de certificación. Típicamente, dichas autoridades mantendrían vínculos con los servidores SFS para los cuales están actuando. Como un ejemplo, consideremos una autoridad de certificación SFS *CA* que ejecuta el servidor SFS de nombre

/sfs/sfs.certsfs.com:kty83pad72qmbna9uefdppioq7053jux

Si suponemos que el cliente ya instaló un vínculo simbólico

/certsfs → */sfs/sfs.certsfs.com:kty83pad72qmbna9uefdppioq7053jux,*

la autoridad de certificación podría utilizar otro vínculo simbólico

/vucs → */sfs/sfs.vu.cs.nl:ag62hty4wior450hdh63u62i4f0kqere*

que apunta al servidor SFS *sfs.vu.cs.nl*. En este caso, un cliente simplemente puede referirse a */certsfs/vucs/home/steen/mbox* sabiendo que está accesando un servidor de archivos cuya clave pública ha sido certificada por la autoridad de certificación *CA*.

Si regresamos al problema de autenticación descentralizada, ahora deberá ser claro que se tienen todos los mecanismos para evitar que Bob deba estar registrado en el servidor de autenticación de Alicia. En cambio, este último servidor puede simplemente ponerse en contacto con el servidor de Bob siempre y cuando éste posea un nombre. Ese nombre ya contiene una clave pública de modo que el servidor de Alicia puede verificar la identidad del servidor de Bob. Después de eso, el servidor de Alicia puede aceptar los privilegios de Bob tal como fueron indicados por Alicia. Como ya mencionamos, los detalles de este esquema se encuentran en Kaminsky y colaboradores (2003).

11.8.3 Sistema de compartimiento de archivos seguros punto a punto

Hasta ahora, hemos analizado sistemas de archivo distribuidos que eran relativamente fáciles de asegurar. Los sistemas tradicionales, o aplican autenticación directa y mecanismos de control de acceso ampliados con comunicación segura, o podemos utilizar autenticación tradicional para implementar un esquema completamente descentralizado. Sin embargo, surgen problemas cuando se trata de sistemas totalmente descentralizados basados en la colaboración, tales como sistemas de compartimiento de archivos de punto a punto.

Búsquedas seguras en sistemas basados en DHT

Existen varios temas que deben ser abordados (Castro y cols., 2002a; y Wallach, 2002). Consideremos los sistemas basados en DHT. En este caso, debemos confiar en operaciones de búsqueda seguras, las que en esencia se reducen a la necesidad de tener una ruta segura. Esto significa que cuando un nodo no defectuoso busca una clave *k*, su solicitud es remitida al nodo responsable de los datos asociados con *k*, o a un nodo que guarde una copia de los datos. La ruta segura requiere que se tomen en cuenta tres aspectos:

1. A los nodos se les asignan identificadores de una forma segura.
2. Las tablas de ruta se mantienen seguras.
3. Las solicitudes de búsqueda son remitidas con seguridad entre nodos.

Cuando a los nodos no se les asigna su identificador con seguridad, se puede enfrentar el problema de que un nodo malicioso pueda asignarse a sí mismo un ID de modo que todas las búsquedas de claves específicas se dirijan a él, o sean remitidas a lo largo de la ruta de la que forma parte. Esta situación se vuelve más compleja cuando los nodos pueden hacer equipo, lo cual, efectivamente, permite que un grupo conforme un enorme "resumidero" para muchas solicitudes de búsqueda. Asimismo, sin la asignación segura de un identificador, un solo nodo también puede ser capaz de autoasignarse *muchos* identificadores, esto se conoce también como **ataque de Sybil** y crea el mismo efecto (Douceur, 2002).

Más general que el ataque de Sybil es un ataque mediante el cual un nodo malicioso controla a tantos nodos vecinos no defectuosos que llega a ser virtualmente imposible corregirlos para

que operen de manera apropiada. Este fenómeno también se conoce como **ataque eclipse**, y se analiza en Singh y colaboradores (2006). Defenderse contra semejante ataque es difícil. Una solución razonable es restringir el número de conexiones entrantes para cada nodo. Así, un atacante sólo puede tener un número limitado de nodos correctos apuntando hacia él. Además, para evitar que un atacante se apropie de todos los vínculos entrantes a nodos correctos, el número de vínculos salientes también deberá limitarse [vea también Singh y colaboradores (2004)]. Problemático en todos los casos es que se requiera de una autoridad centralizada para entregar identificadores de nodo. Desde luego, tal autoridad atenta contra la naturaleza descentralizada de los sistemas punto a punto.

Cuando las tablas de ruta se pueden llenar con nodos alternativos, como a menudo es el caso cuando se optimiza para proximidad de red, un atacante puede convencer con facilidad a un nodo para que apunte a nodos maliciosos. Observe que este problema no ocurre cuando existen fuertes limitaciones sobre el llenado de tablas de ruta, como en el caso del sistema de cuerdas. La solución, por consiguiente, es mezclar la selección de alternativas con un llenado más restringido de las tablas [cuyos detalles se describen en Castro y cols. (2002a)].

Por último, para defenderse contra ataques de remisión de mensajes, un nodo puede simplemente remitir mensajes a lo largo de varias rutas. Una forma de hacerlo es iniciar una búsqueda desde diferentes nodos origen.

Almacenamiento seguro colaborativo

No obstante, el simple hecho de que se requiere que los nodos colaboren provoca más problemas. Por ejemplo, la colaboración puede dictar que los nodos deberán ofrecer aproximadamente la misma cantidad de almacenamiento que utilizan de otros. La aplicación de esta política puede resultar bastante conflictiva. Una solución es aplicar un intercambio de almacenamiento seguro, como para Samsara, según describen Cox y Noble (2003).

La idea es bastante simple: cuando un servidor *P* desea guardar uno de sus archivos *f* en otro servidor *Q*, pone a disposición un espacio de almacenamiento de tamaño igual al de *f* y reserva ese espacio exclusivamente para *Q*. En otros términos, ahora *Q* tiene una **reclamación** pendiente en *A*, como se muestra en la figura 11-33.

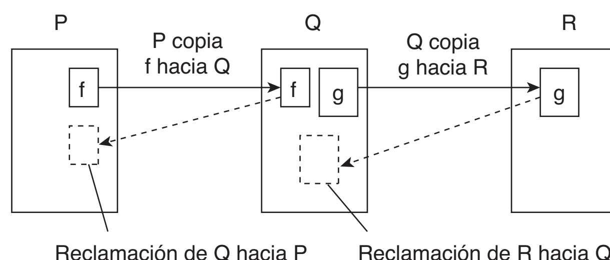


Figura 11-33. Principio de reclamación de espacio de almacenamiento en el sistema Samsara de igual a igual.

Para hacer que este esquema funcione, cada participante reserva cierta cantidad de almacenamiento y lo divide en fragmentos del mismo tamaño. Cada fragmento consta de datos que no pueden comprimirse. En Samsara, el fragmento c_i consiste en un valor disperso (hash) h_i de 160 bits, calculado a lo largo de una frase de pase secreta *W* concatenada con el número *i*. Supongamos ahora que las reclamaciones se hacen en unidades de 256 bytes. En ese caso, la primera reclamación se calcula tomando los primeros 12 fragmentos junto con los primeros 16 bytes del siguiente fragmento. Estos fragmentos se concatenan y cifran con una clave privada *K*. En general, la reclamación C_j se calcula como

$$C_j = K(h_k, h_{k+1}, \ldots, h_{k+11}, h_{k+12}[0], \ldots, h_{k+12}[15])$$

donde $k = j \times 13$. Siempre que *P* desea utilizar el almacenamiento disponible en *Q*, *Q* regresa un conjunto de reclamaciones que *P* está obligado a guardar entonces. Desde luego, *Q* nunca tiene que guardar sus propias reclamaciones. En cambio, puede calcular cuándo se requieren.

El truco ahora es que, de vez en cuando, tal vez *Q* desee verificar si *P* sigue guardando sus reclamaciones. Si *P* no puede demostrar que sí lo está haciendo, *Q* simplemente desecha los datos de *P*. Una forma brusca de permitir que *P* demuestre que aún tiene las reclamaciones es regresando copias a *Q*. Desde luego, esto desperdiciará una gran cantidad de ancho de banda. Supongamos que *Q* había entregado las reclamaciones $C_{j_1}, \ldots, C_{j_k}$ a *P*. En ese caso, *Q* pasa una cadena *d* de 160 bits a *P*, y le solicita calcular la dispersión (hash) d_1 de *d* de 160 bits concatenado con C_{j_1}. Esta dispersión se concatena entonces con C_{j_2} para producir un valor disperso d_2 y así sucesivamente. Al final, *P* sólo tiene que regresar d_n para demostrar que aún tiene las reclamaciones.

Desde luego, tal vez *Q* desee replicar sus archivos en otro nodo, por ejemplo, en *R*. Para hacerlo, tendrá que conservar las reclamaciones para *R*. Sin embargo, si a *Q* se le está acabando su espacio de almacenamiento, pero ha reclamado almacenamiento en *P*, puede decidir pasar las reclamaciones a *R*. Este principio funciona como sigue.

Supongamos que *P* conserva una reclamación C_Q para *Q*, y se supone que *Q* conserva una reclamación C_R para *R*. Como no existe ninguna restricción sobre qué puede guardar *Q* en *P*, *Q* muy bien podría decidir guardar C_R en *P*. Entonces, siempre que *R* desee comprobar si *Q* aún conserva su reclamación, *Q* pasará un valor *d* a *Q* y le pedirá que calcule el hash de *d* concatenado con C_R. Para hacerlo, *Q* simplemente pasa *d* a *P* y le pide calcular el hash y regresar el resultado a *R*. Si resulta que *P* ya no conserva la reclamación, *Q* será castigado por *R*, y *Q*, a su vez, puede castigar a *P* eliminando los datos guardados.

11.9 RESUMEN

Los sistemas de archivo distribuidos conforman un importante paradigma para la construcción de sistemas distribuidos. En general, están organizados de acuerdo con el modelo cliente-servidor, con almacenamiento en la memoria caché del lado del cliente y soporte de replicación de servidor para satisfacer requerimientos de escalabilidad. Además, el almacenamiento en caché y la replicación se requieren para alcanzar una alta escalabilidad. Más recientemente, emergieron arquitecturas simétricas tales como las de los sistemas de compartimiento de archivos punto a punto. En estos casos, un tema importante es saber si archivos completos o bloques de datos están distribuidos.

En lugar de construir un sistema de archivo directamente encima de la capa de transporte, es práctica común asumir la existencia de una capa RPC, de modo que todas las operaciones simplemente puedan ser expresadas como RPC a un servidor de archivos en lugar de tener que utilizar operaciones primitivas de transferencia de mensajes. Se han desarrollado algunas variantes de la RPC, tales como MultiRPC en Coda, las cuales permiten llamar a varios servidores puestos en paralelo.

Lo que hace diferentes a los sistemas de archivo distribuidos de los sistemas de archivo no distribuidos es la semántica de compartimiento de archivos. Idealmente, un sistema de archivos permite que un cliente siempre lea los datos más recientemente escritos en un archivo. Estas semánticas UNIX de compartimiento de archivos son muy difíciles de implementar con eficiencia en un sistema distribuido. NFS soporta una forma más débil conocida como semántica de sesión, mediante la cual la versión final de un archivo es determinada por el último cliente que cierra el archivo, el que previamente había abierto para escribir. En Coda, el compartimiento de archivos se apega a la semántica transaccional en el sentido de que los clientes que leen sólo podrán ver las actualizaciones más recientes si reabren un archivo. La semántica transaccional implementada en Coda no cubre todas las propiedades ACID de las transacciones regulares. En el caso de que un servidor de archivos permanezca en control de todas las operaciones, puede ser provista semántica UNIX real, aunque entonces la escalabilidad resultará ser un problema. En todos los casos, es necesario permitir actualizaciones concurrentes en archivos, ello pone en juego esquemas de reservación y bloqueo relativamente confusos.

Para lograr un desempeño aceptable, los sistemas de archivo distribuidos permiten generalmente que los clientes guarden en la memoria caché un archivo completo. Este método de guardar en caché archivos completos es soportado, por ejemplo, en NFS, aunque también es posible guardar sólo fragmentos muy grandes de un archivo. Una vez que el archivo ha sido abierto y (en parte) transferido al cliente, todas las operaciones se realizan localmente. Las actualizaciones son devueltas al servidor cuando el archivo se cierra otra vez.

La replicación también desempeña un rol importante en sistemas punto a punto, aunque las cosas se simplifican demasiado porque los archivos son generalmente de sólo lectura. En estos sistemas es más importante tratar de alcanzar un equilibrio de carga aceptable, ya que los esquemas de replicación ingenuos pueden conducir fácilmente a puntos conflictivos que guardan muchos archivos y que, por tanto, pueden convertirse en cuellos de botella potenciales.

La tolerancia a fallas normalmente se maneja con métodos tradicionales. Sin embargo, también es posible construir sistemas de archivo capaces de manejar fallas bizantinas, aun cuando el sistema en su totalidad funcione en internet. En este caso, suponiendo tiempos fuera razonables e iniciando grupos de servidores nuevos (posiblemente basados en una falsa detección de fallas), se pueden construir soluciones prácticas. Para sistemas de archivo distribuidos, principalmente, se deberá considerar aplicar técnicas de codificación de borradura para reducir el factor de replicación total cuando sólo se busca una alta disponibilidad.

La seguridad es de primordial importancia para cualquier sistema distribuido, incluidos sistemas de archivo. NFS difícilmente proporciona mecanismos de seguridad por sí mismo, pero en cambio implementa interfaces estandarizadas que permiten utilizar los distintos sistemas de seguridad existentes, tal como, por ejemplo, Kerberos. SFS es diferente en el sentido de que permite que los nombres de archivo incluyan información sobre la clave pública del servidor de archivos. Este método simplifica la gestión de claves en sistemas implementados a gran escala. En realidad, el SFS

distribuye una clave incluyéndola en el nombre de un archivo. El SFS puede ser utilizado para implementar un esquema de autenticación descentralizado. Lograr seguridad en sistemas de compartimiento de archivos punto a punto es difícil, en parte debido a la supuesta naturaleza colaborativa donde los nodos siempre tenderán a actuar egoístamente. Además, hacer que las búsquedas sean seguras resulta ser un problema difícil que en realidad requiere de la intervención de una autoridad central para entregar identificadores de nodo.

PROBLEMAS

1. ¿Se requiere que un servidor de archivos que implementa el NFS versión 3 sea sin estado?

2. Explique si el NFS tiene que ser considerado o no como un sistema de archivo distribuido.

3. A pesar de que el GFS crece bien, se podría argumentar que el maestro continúa siendo un cuello de botella. ¿Cuál sería una alternativa razonable para reemplazarlo?

4. El uso de los efectos colaterales del RPC2 es conveniente para flujos de datos continuos. Proporcione otro ejemplo en el cual tenga sentido utilizar un protocolo específico de una aplicación al lado de RPC.

5. El NFS no proporciona un espacio de nombre compartido global. ¿Existe alguna forma de imitarlo tal como un espacio de nombre?

6. Proporcione una extensión simple de la operación lookup de NFS que permita la búsqueda iterativa de un nombre en combinación con un servidor que exporta directorios montado desde otro servidor.

7. En sistemas operativos basados en UNIX, la apertura de un archivo por medio de un manejador sólo puede hacerse en el kernel. Proporcione una posible implementación de un manejador de archivo NFS para un servidor NFS a nivel de usuario para un sistema UNIX.

8. El uso de un automontador que instale vínculos simbólicos, tal como se describe en el texto, vuelve más difícil ocultar el hecho de que el montaje es transparente. ¿Por qué?

9. Suponga que el estado de denegación actual de un archivo en NFS es *WRITE*. ¿Es posible que otro cliente pueda abrir por primera vez con éxito el archivo y luego solicitar un bloqueo de escritura?

10. Si tomamos en cuenta el tema de la coherencia del caché analizado en el capítulo 7, ¿qué clase de protocolo de coherencia de caché implementa el NFS?

11. ¿Implementa el NFS la consistencia de entradas?

12. Estipulamos que el NFS implementa el modelo de acceso remoto al manejo de archivos. ¿Se puede argumentar que también soporta el modelo de carga y descarga? Explique por qué.

13. En el NFS, los atributos se guardan en la memoria caché mediante una política de coherencia del caché de escritura. ¿Es necesario remitir todos los cambios de atributo de inmediato?

14. ¿Qué semántica de invocación proporciona el RPC2 en la presencia de fallas?

15. Explique cómo resuelve Coda los conflictos de lectura y escritura en un archivo compartido entre múltiples lectores y un solo escritor.

16. Con el uso de nombres de ruta autocertificantes, ¿un cliente siempre puede estar seguro de que se está comunicando con un servidor no malicioso?

17. (**Tarea para el laboratorio.**) Una de las formas más fáciles de construir un sistema distribuido basado en UNIX es acoplar varias máquinas por medio de un NFS. Para realizar esta tarea, usted tiene que conectar dos sistemas de archivo en computadoras diferentes por medio del NFS. En particular, instale un servidor NFS en una máquina de tal forma que varias partes de su sistema de archivo se monten automáticamente cuando la primera máquina se inicie.

18. (**Tarea para el laboratorio.**) Para integrar máquinas basadas en UNIX con clientes Windows, se pueden utilizar servidores Samba. Amplíe la tarea previa poniendo a la disposición de un cliente Windows un sistema basado en UNIX, instalando y configurando un servidor Samba. Al mismo tiempo, el sistema de archivos deberá permanecer accesible a través del NFS.

12

SISTEMAS DISTRIBUIDOS BASADOS EN LA WEB

La Red mundial (**WWW**, del inglés *World Wide Web*) puede ser vista como un enorme sistema distribuido compuesto a partir de millones de clientes y servidores para poder acceder a documentos vinculados. Los servidores mantienen conjuntos de documentos, en tanto que los clientes proporcionan a los usuarios una interfaz de fácil uso para presentar y acceder a tales documentos.

La web se inició como un proyecto en CERN, el Laboratorio Europeo de Física de Partículas con sede en Ginebra, para permitir que su gran y geográficamente disperso grupo de investigadores tuviera acceso a documentos compartidos mediante un simple sistema de hipertexto. Un documento podría ser cualquier cosa que pudiera desplegarse en la terminal de un usuario en una computadora, tal como notas personales, informes, cifras, planos (*blueprints*), dibujos y así sucesivamente. Vinculando los documentos entre sí, fue fácil integrarlos desde diferentes proyectos en un nuevo documento sin la necesidad de realizar cambios centralizados. Lo único que se requería era construir un documento que proporcionara vínculos a otros documentos pertinentes [vea también Berners-Lee y cols. (1994)].

La web creció gradual y lentamente con sitios que no eran de física de alta energía, pero su popularidad se incrementó en forma impresionante cuando las interfaces de usuario gráficas estuvieron disponibles, principalmente Mosaic (Vetter y cols., 1994). Mosaic proporcionó una interfaz de fácil uso para presentar y acceder a documentos simplemente con hacer clic en un botón del ratón. Se buscaba un documento en un servidor, se transfería al cliente y se presentaba en la pantalla. Para un usuario, no había ninguna diferencia conceptual entre un documento guardado localmente o en otra parte del planeta. En este sentido, la distribución era transparente.

A partir de 1994, el World Wide Web Consortium (Consorcio de la red mundial) inició los desarrollos, una colaboración entre el CERN y el Massachusetts Institute of Technology. Este consorcio es responsable de estandarizar los protocolos, mejorar la interoperabilidad, y dar mayor realce a las capacidades de la web. Además, se puede advertir que por fuera de este consorcio se están dando muchos desarrollos nuevos que no siempre conducen a la compatibilidad deseada. Por ahora, la web no es más que un simple sistema basado en documentos. De manera notoria, con la introducción de servicios se está viendo el surgimiento de un enorme sistema de distribución donde se están utilizando, componiendo y ofreciendo **servicios** a cualquier usuario o máquina que sea capaz de utilizarlos.

En este capítulo examinaremos este sistema omnipresente de rápido crecimiento. Considerando que la web es muy reciente y que mucho ha cambiado en corto tiempo, nuestra descripción podría ser sólo una imagen instantánea de su estado actual. Sin embargo, como veremos, muchos conceptos que fundamentan la tecnología de la web están basados en los principios abordados en la primera parte de este libro. También veremos que para muchos conceptos, aún existe mucho por mejorar.

12.1 ARQUITECTURA

La arquitectura de los sistemas distribuidos basados en la web no es fundamentalmente distinta de la de otros sistemas distribuidos. Sin embargo, es interesante ver cómo ha evolucionado la idea inicial de dar soporte a documentos distribuidos desde su lanzamiento en los años de 1990. Los documentos se transformaron de ser puramente estáticos y pasivos a dinámicamente generados que contienen toda clase de elementos activos. Además, en años recientes, muchas organizaciones han comenzado a soportar servicios en lugar de sólo documentos. A continuación analizamos los impactos arquitectónicos de estos cambios.

12.1.1 Sistemas tradicionales basados en la web

A diferencia de muchos sistemas distribuidos presentados hasta ahora, los sistemas distribuidos basados en la web son relativamente nuevos. En este sentido, es un tanto difícil hablar sobre sistemas tradicionales basados en la web, aunque existe una clara distinción entre los sistemas que estuvieron disponibles al principio y los utilizados en la actualidad.

Muchos sistemas basados en la web están organizados como arquitecturas cliente-servidor relativamente simples. La parte central de un sitio web está conformada por un proceso que tiene acceso a un sistema de archivo local que guarda documentos. El modo más simple de referirse a un documento es por medio de una referencia llamada **localizador uniforme de recursos** (**URL**, por sus siglas en inglés). El URL especifica la localización de un documento, a menudo incluyendo un nombre DNS de su servidor asociado junto con el nombre del archivo mediante el cual el servidor puede buscar el documento en su sistema de archivo local. Además, un URL especifica el protocolo a nivel de aplicación para transferir el documento a través de la red. Existen varios protocolos diferentes disponibles, como se explica a continuación.

Un cliente interactúa con servidores web a través de una aplicación especial conocida como **navegador**. Un navegador es responsable de desplegar apropiadamente un documento. También, un navegador acepta entradas de un usuario en su mayor parte para permitirle seleccionar una referencia a otro documento, el cual entonces el navegador busca y despliega. La comunicación entre un navegador y un servidor web se ha estandarizado: ambos se adhieren al **protocolo de transferencia de hipertexto** (**HTTP**, por sus siglas en inglés), el que analizaremos a continuación. Esto conduce a la organización mostrada en la figura 12-1.

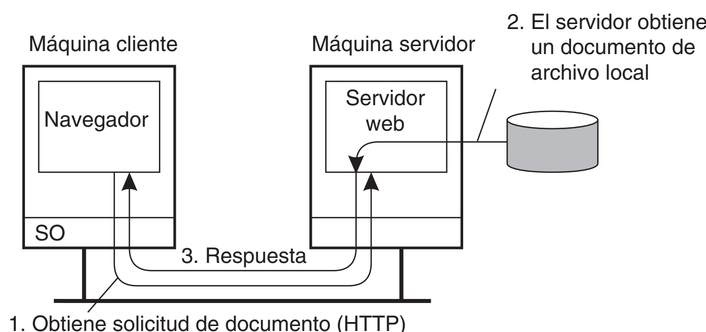


Figura 12-1. Organización total de un sitio web tradicional.

La web ha evolucionado en forma considerable desde su presentación. Por el momento, existe una inmensa variedad de métodos y herramientas que producen información que puede ser procesada por clientes web y servidores web. A continuación, examinaremos con detalle la manera en que actúa la web como un sistema distribuido. Sin embargo, pasamos por alto la mayoría de las herramientas y métodos utilizados para construir documentos web, pues con frecuencia no tienen una relación directa con la naturaleza distribuida de la web. Una buena introducción sobre cómo construir aplicaciones basadas en la web se encuentra en Sebesta (2006).

Documentos web

Para la web es fundamental que, virtualmente, toda la información llega en la forma de un documento. El concepto de un documento tiene que ser tomado en su sentido más amplio; no sólo puede contener texto simple, sino también incluir diversas clases de características dinámicas tales como audio, video y animaciones, y así sucesivamente. En muchos casos, se requieren aplicaciones de ayuda especiales para hacer que un documento "cobre vida". Tales interpretadores típicamente estarán integrados al navegador de usuario.

La mayoría de los documentos pueden dividirse aproximadamente en dos partes: una parte principal que al final actúa como plantilla para la segunda parte, la cual se compone de muchos bits y piezas diferentes que en conjunto constituyen el documento desplegado en un navegador. La parte principal, en general, se escribe en **lenguaje de marcas**, muy similar al tipo de lenguajes

utilizados en sistemas de proceso de palabras. El lenguaje de marcas que se utiliza más ampliamente en la web, el cual es acrónimo para **lenguaje de marcas de hipertexto** (**HTML**, por sus siglas en inglés). Como su nombre lo indica, el HTML permite insertar vínculos a otros documentos. Cuando se activan en un navegador, el documento referido será tomado de su servidor asociado.

Otro lenguaje de marcado cada vez más importante es el **lenguaje de marcado extensible** (**XML**, por sus siglas en inglés), el cual, como su nombre sugiere, es más flexible al definir cómo deberá verse un documento. La diferencia principal entre el HTML y el XML, es que éste incluye las definiciones de los elementos que marcan un documento. En otros términos, es un lenguaje de metamarcado. Este método permite una gran flexibilidad cuando se trata de especificar con exactitud cómo deberá verse un documento: no existe la necesidad de apegarse a un solo modelo en la forma que lo dicta un lenguaje de marcado fijo tal como el HTML.

El HTML y el XML también incluyen toda clase de señalizaciones que se refieren a **documentos embebidos**, es decir, referencias a archivos que deberán ser incluidos para que el documento esté completo. Se puede argumentar que los documentos embebidos transforman un documento web en algo activo. En especial, cuando se considera que un documento embebido puede ser un programa completo ejecutado al vuelo como parte del despliegue de información, no es difícil imaginar lo que se puede hacer.

Los documentos insertados vienen en toda clase de sabores. Esto inmediatamente trae a colación el tema de cómo se pueden equipar los navegadores para manejar los diferentes tipos de formatos de archivo y las maneras de interpretarlos. En esencia, se requieren sólo dos cosas: una forma de especificar el tipo de documento embebido y un modo de permitir que un navegador maneje datos de un tipo específico.

Cada documento (embebido) lleva un **tipo MIME** asociado. MIME proviene del inglés *Multipurpose Internet Mail Exchange* —significa **intercambio de correo en internet para múltiples usos**— y, como su nombre sugiere, originalmente se desarrolló para que informara sobre el contenido del cuerpo de un mensaje enviado como parte del correo electrónico. MIME distingue varios tipos de contenidos de mensaje. Estos tipos también se utilizan en la WWW, aunque la estandarización resulta difícil con los nuevos formatos de datos que aparecen casi a diario.

MIME distingue entre tipos de alto nivel y subtipos. Algunos tipos de alto nivel se muestran en la figura 12-2 e incluyen tipos de texto, imagen, audio y video. Existe un tipo de *aplicación* especial para indicar que el documento contiene datos relacionados con una aplicación específica. En la práctica, sólo esa aplicación será capaz de transformar el documento en algo que pueda ser entendido por un ser humano.

El tipo de *múltiples partes* se utiliza para documentos compuestos, es decir, documentos constituidos por varias partes donde cada parte desempeña su propio tipo de alto nivel asociado.

Por cada tipo de alto nivel, puede haber varios subtipos disponibles, de los cuales algunos también se muestran en la figura 12-2. El tipo de un documento se representa entonces en la forma de una combinación de un tipo de alto nivel y un subtipo, tal como, por ejemplo, la *aplicación/PDF*. En este caso, se espera que sea necesario utilizar una aplicación distinta para procesar el documento, la cual está representada en PDF. Muchos subtipos son experimentales, es decir, utilizan un formato especial que requiere su propia aplicación del lado del usuario. En la práctica, es el servidor web el que la proporcionará, o como un programa aparte que funcionará al lado

Tipo	Subtipo	Descripción
Texto	Común	Texto sin formato
	HTML	Texto que incluye comandos de marcado HTML
	XML	Texto que incluye comandos de marcado XML
Imagen	GIF	Imagen fija en formato GIF
	JPEG	Imagen fija en formato JPEG
Audio	Básico	Audio, PCM de 8 bits muestreado a 8000 Hz
	Tonos	Un tono audible específico
Video	MPEG	Cine en formato MPEG
	Apuntador	Representación de un dispositivo apuntador para presentaciones
Aplicación	Flujo de octetos	Una secuencia de bytes no interpretada
	Postscript	Un documento imprimible en postscript
	PDF	Un documento imprimible en PDF
Multipartes	Combinadas	Partes independientes en el orden especificado
	Paralelas	Partes que deben ser vistas al mismo tiempo

Figura 12-2. Seis tipos MIME de nivel superior y algunos subtipos comunes.

del navegador, o como un **plug-in** (**componente de software adicional**) que pueda ser instalado como parte del navegador.

Esta variedad (cambiante) de tipos de documentos obliga a que los navegadores sean extensibles. Con este objeto, se ha llevado a cabo cierta estandarización para permitir que los plug in se adhieran a ciertas interfaces e integrarlos con facilidad al navegador. Cuando ciertos tipos se vuelven lo suficientemente populares, a menudo vienen incorporados a los navegadores o a sus actualizaciones. Más adelante volveremos al tema, cuando abordemos el software cliente-servidor.

Arquitecturas de varios niveles

La combinación de HTML (o cualquier otro tipo de lenguaje de marcado tal como XML) con la posibilidad de programación por medio de guiones (*scripting*) resulta en un medio poderoso para expresar documentos. Sin embargo, apenas si analizamos dónde son procesados en realidad los documentos, y qué clase de procesamiento tiene lugar. La WWW se inició como el sistema cliente-servidor de dos niveles relativamente simple mostrado en la figura 12-1. Por el momento, diremos que esta arquitectura simple se amplió con numerosos componentes para soportar el tipo avanzado de documentos que se acaban de describir.

Una de las primeras mejoras de la arquitectura básica fue soportar la interacción de un usuario por medio de la **interfaz de compuerta común**, o simplemente **CGI** (por sus siglas en inglés). La CGI define una forma estándar mediante la cual un servidor web es capaz de ejecutar un programa tomando los datos de un usuario como entrada. En general, los datos de usuario provienen de una forma HTML; ésta especifica el programa a ser ejecutado del lado del servidor, junto con

valores de parámetro establecidos por el usuario. Una vez completada la forma, el nombre del programa y los valores de parámetro reunidos son enviados al servidor, como se muestra en la figura 12-3.

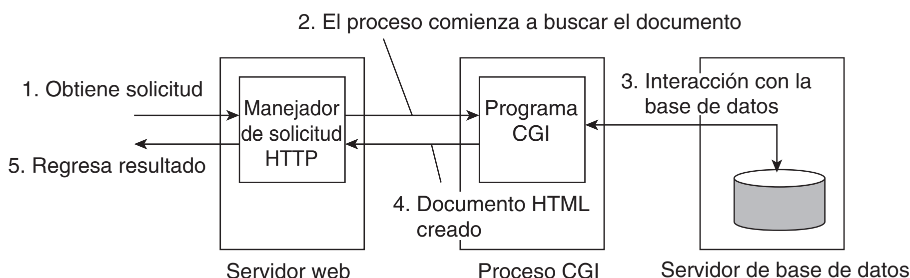


Figura 12-3. Principio de la utilización de programas CGI del lado del servidor.

Cuando el servidor ve la solicitud, inicia el programa nombrado en la solicitud y transfiere los valores de parámetro. En ese momento, el programa simplemente realiza su trabajo, y por lo general regresa los resultados en la forma de un documento devuelto al navegador de usuario para que lo despliegue.

Los programas CGI pueden ser tan complejos como lo desee el desarrollador. Por ejemplo, como se muestra en la figura 12-3, muchos programas operan con una base de datos local con respecto al servidor web. Después de procesar los datos, el programa genera un documento HTML y lo regresa al servidor. El servidor luego se lo envía al cliente. Una observación interesante es que ante el servidor, aparece como si éste le estuviera pidiendo al programa CGI que busque un documento. En otros términos, el servidor no hace más que delegar la búsqueda de un documento a un programa externo.

La tarea principal de un servidor era manejar las solicitudes de los clientes buscando simplemente documentos. Con programas CGI, la búsqueda de un documento podría ser delegada de tal forma que el servidor no supiera si un documento había sido generado al vuelo, o en realidad leído desde un sistema de archivo local. Observe que acabamos de describir la organización en dos niveles de un software del lado del servidor.

Sin embargo, los servidores actuales hacen mucho más que sólo buscar documentos. Una de las mejoras más importantes es que también pueden procesar un documento antes de pasarlo al cliente. En particular, un documento puede contener un **script del lado del servidor**, el cual es ejecutado por el servidor cuando el documento ha sido buscado localmente. El resultado de ejecutar un script es enviado al cliente junto con el resto del documento. El script en sí no es enviado. En otros términos, el uso de un script del lado del servidor cambia un documento esencialmente al reemplazar el script con los resultados de su ejecución.

Como el procesamiento del lado del servidor de documentos web cada vez requiere más flexibilidad, no sorprende que ahora muchos sitios web estén organizados como una arquitectura de tres niveles compuesta a partir de un servidor web, un servidor de aplicaciones, y una base de datos.

El servidor web es el tradicional de antes; el servidor de aplicaciones ejecuta toda clase de programas que pueden o no acceder al tercer nivel, compuesto por una base de datos. Por ejemplo, un servidor puede aceptar la petición de un cliente, buscar en su base de datos productos iguales, y luego construir una página web que contenga una lista de los productos encontrados donde se pueda hacer clic. En muchos casos, el servidor es responsable de ejecutar programas Java, llamados **servlets**, que mantienen cosas como carritos de supermercado, implementan recomendaciones, mantienen listas de artículos y así sucesivamente.

Esta organización en tres niveles presenta un problema, sin embargo: reduce el desempeño. Aunque desde un punto de vista arquitectónico tiene sentido distinguir tres niveles, la práctica indica que el servidor de aplicaciones y la base de datos son cuellos de botella potenciales. En forma importante, tratar de mejorar el desempeño de una base de datos puede resultar en un desagradable problema. Más adelante regresaremos al tema, cuando analicemos el almacenamiento en la memoria caché y la replicación como soluciones a problemas de desempeño.

12.1.2 Servicios web

Hasta ahora, hemos asumido en forma implícita que el software del lado del cliente de un sistema basado en la web se compone de un navegador que actúa como interfaz para un usuario. Esta suposición ya no es universalmente cierta. Existe un grupo de rápido crecimiento de sistemas basados en la web que ofrecen servicios generales a aplicaciones remotas sin interacciones intermedias de usuarios finales. Esta organización conduce al concepto de **servicios web** (Alonso y cols., 2004).

Fundamentos de los servicios web

Expresado en forma simple, un servicio web no es nada más que un servicio tradicional (por ejemplo, un servicio de asignación de nombres, un servicio de reporte meteorológico, un proveedor electrónico, etc.) puesto a la disposición de todo mundo en internet. Lo que hace especial al servicio web es que se apega a un conjunto de estándares que le permiten ser *descubierto* y accesado a través de la red por aplicaciones cliente que también se apegan a dichos estándares. No sorprende entonces que dichos estándares conformen el núcleo de la arquitectura de los servicios web [vea también Booth y cols. (2004)].

El principio de proporcionar y utilizar un servicio web es bastante simple y se muestra en la figura 12-4. La idea básica es que alguna aplicación cliente puede solicitar los servicios provistos por una aplicación servidor. La estandarización ocurre con respecto a cómo son descritos los servicios de tal forma que puedan ser buscados por una aplicación cliente. Además, se tiene que garantizar que la solicitud de servicios siga las reglas establecidas por la aplicación servidor. Observe que este principio no es diferente de lo que se requiere para invocar un procedimiento remoto.

Un componente importante en la arquitectura de servicios web está formado por un servicio de directorio que guarda descripciones del servicio. Este servicio se adhiere al estándar **descripción, descubrimiento e integración universales** (**UDDI**, por sus siglas en inglés). Como su nombre lo indica, UDDI prescribe el plan general de una base de datos que contiene las descripciones del servicio que permitirá a los clientes web buscar los servicios pertinentes.

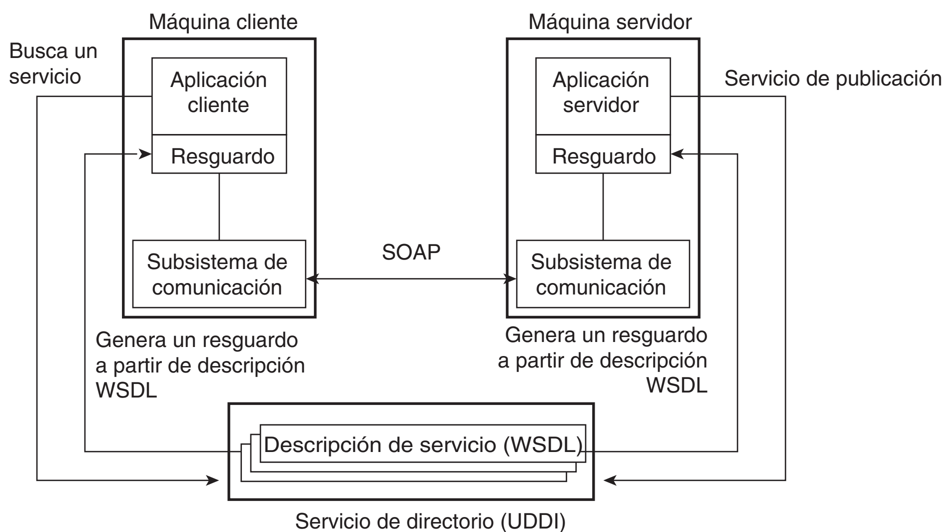


Figura 12-4. Principio de un servicio web.

Los servicios se describen por medio del **lenguaje de definición de servicios web** (**WSDL**, por sus siglas en inglés), el cual es un lenguaje formal muy parecido a los lenguajes de definición de interfaz utilizados para soportar la comunicación basada en RPC. Una descripción del WSDL contiene definiciones precisas de las interfaces provistas por un servicio, es decir, especificación de procedimiento, tipos de datos, localización (lógica) de servicios, etc. Un importante aspecto en la descripción de un WSDL es que puede ser transformado automáticamente en Resguardos del lado del cliente y en resguardos del lado del servidor, de nuevo, como en la generación de resguardos en sistemas basados en RPC comunes.

Por último, un elemento central de un servicio web es la especificación de cómo ocurre la comunicación. Con esta finalidad, se utiliza el **protocolo de acceso a un objeto simple** (**SOAP**, por sus siglas en inglés), el cual es, en esencia, una estructura donde una gran parte de la comunicación entre dos procesos puede ser estandarizada. A continuación analizaremos más a fondo el SOAP, y también veremos que llamar al marco de trabajo simple en realidad no se justifica.

Coordinación y composición de los servicios web

La arquitectura descrita hasta ahora es relativamente sencilla: un servicio simplemente se implementa por medio de una aplicación y su invocación ocurre de acuerdo con un estándar específico. Desde luego, la aplicación en sí puede ser compleja y, de hecho, sus componentes pueden estar completamente distribuidos a través de una red de área local. En esos casos, es muy probable que el servicio web se implemente por medio de un proxy o un demonio interno que interactúe con los

diversos componentes constitutivos de la aplicación distribuida. En ese caso, todos los principios analizados hasta ahora pueden ser aplicados con facilidad, como ya vimos.

En el modelo presentado hasta aquí, se ofrece un servicio web en la forma de una sola invocación. En la práctica, deben existir estructuras de invocación mucho más complejas antes de poder considerar que se completó un servicio. Por ejemplo, consideremos una librería electrónica. Pedir un libro requiere seleccionarlo, pagarlo, y garantizar su entrega. Desde una perspectiva de servicio, el servicio en sí deberá ser modelado como una transacción compuesta de múltiples pasos que deben ser realizados en un orden específico. En otros términos, se trata de un **servicio complejo** integrado por varios servicios básicos.

La complejidad se incrementa cuando consideramos servicios web ofrecidos al combinar servicios web de diferentes proveedores. Un ejemplo típico es imaginar una tienda basada en la web. La mayoría de las tiendas se componen aproximadamente de tres partes: una parte fija mediante la cual un cliente selecciona los artículos que requiere, una segunda parte que maneja el pago de dichos artículos, y una tercera parte encargada de enviar y posteriormente rastrear los artículos. Para establecer una tienda como esa, es posible que un proveedor desee utilizar un servicio de banco electrónico que pueda manejar los pagos, pero también un servicio de entrega especial encargado de enviar los artículos. De este modo, un proveedor puede concentrarse en su negocio central, que es la oferta de artículos.

En estos escenarios es importante que un cliente vea un servicio coherente: es decir, una tienda donde pueda seleccionar, pagar, y confiar en una entrega apropiada. Sin embargo, internamente necesitamos tratar con una situación en la que tal vez tres organizaciones diferentes tienen que actuar en forma coordinada. El apoyo apropiado de **servicios compuestos** como esos conforma un elemento esencial de los servicios web. Se presentan por lo menos dos clases de problemas que debemos resolver. En primer lugar, ¿cómo ocurre la coordinación entre servicios web de, posiblemente, diferentes organizaciones? En segundo lugar, ¿cómo se pueden implementar con facilidad los servicios?

La coordinación entre servicios web se logra mediante **protocolos de coordinación**. Un protocolo de coordinación prescribe los diversos pasos a emprender para que el servicio (compuesto) tenga éxito. El tema, desde luego, es *hacer* que las partes que intervienen en un protocolo como ese sigan los pasos correctos en el momento apropiado. Existen varias formas de lograrlo; la forma más simple es que un solo coordinador controle los mensajes intercambiados entre las partes participantes.

Sin embargo, aunque existen varias soluciones, desde la perspectiva de servicios web es importante estandarizar las funciones comunes en los protocolos de coordinación. Por lo menos, es fundamental que cuando una parte desee participar en un protocolo específico, sepa con qué otro(s) proceso(s) debe comunicarse. Además, muy bien puede suceder que un proceso intervenga en múltiples protocolos de coordinación al mismo tiempo. Por consiguiente, también es importante identificar la instancia de un protocolo. Por último, un proceso deberá saber qué rol va a desempeñar.

Estos temas están estandarizados en lo que se conoce como **coordinación de servicios web** (Frend y cols., 2005). Desde un punto de vista arquitectónico, define un servicio aparte de manejo de protocolos de coordinación. La coordinación de un protocolo forma parte de este servicio. Los

procesos pueden autoregistrarse como participantes en la coordinación de modo que sus iguales los conozcan.

Para concretar, consideremos un servicio de coordinación para variantes del protocolo de dos fases (2PC) analizado en el capítulo 8. La idea completa es que tal servicio sería implementado por el coordinador para varias instancias de protocolo. Una implementación evidente es que un solo proceso desempeña el rol de coordinador de múltiples instancias de protocolo. Una alternativa es que hace que cada coordinador sea implementado por un hilo distinto.

Un proceso puede solicitar la activación de un protocolo específico. En ese momento, en esencia, recibirá un identificador que puede transferir a otros procesos para que se registren como participantes en la instancia de protocolo recién creada. Desde luego, se requerirá que todos los procesos participantes implementen las interfaces específicas del protocolo que el servicio de coordinación está soportando. Una vez que todos los participantes se han registrado, el coordinador puede enviarles los mensajes *VOTE_REQUEST*, *COMMIT* y otros mensajes que formen parte del protocolo 2PC, cuando se requiera.

No es difícil advertir que debido a la comunidad en, por ejemplo, protocolos 2PC, la estandarización de interfaces y mensajes para intercambio se volverá mucho más fácil para componer y coordinar los servicios web. El trabajo en sí que debe realizarse no es muy difícil. A este respecto, el valor agregado de un servicio de coordinación deberá buscarse por completo en la estandarización.

Queda claro que un servicio de coordinación ya ofrece los medios necesarios para componer un servicio web con otros servicios. Existe sólo un problema potencial: la composición del servicio, es pública. En muchos casos, ésta no es una propiedad deseable, ya que le permitiría a cualquier competidor establecer con exactitud el mismo servicio compuesto. Lo que se requiere, por consiguiente, son medios para establecer coordinadores privados. Aquí no abordaremos los detalles, ya que no afectan los principios de composición de servicio en sistemas basados en la web. También, este tipo de composición sigue siendo muy utilizado (y continuará siéndolo por mucho tiempo). El lector interesado deberá remitirse a (Alonso y cols., 2004).

12.2 PROCESOS

Ahora abordaremos los procesos más importantes utilizados en sistemas basados en la web y su organización interna.

12.2.1 Clientes

El cliente web más importante es una pieza de software llamada **navegador web**, el cual permite a un usuario navegar a través de páginas web trayéndolas de servidores y, posteriormente, desplegándolas en la pantalla del usuario. Un navegador proporciona, típicamente, una interfaz mediante la cual aparecen hipervínculos de tal forma que el usuario pueda seleccionarlos con facilidad mediante un solo clic del ratón.

Los navegadores web eran programas simples, pero eso fue hace mucho. Lógicamente, constan de varios componentes, mostrados en la figura 12-5 [vea también Grosskurth y Godfrey (2005)].

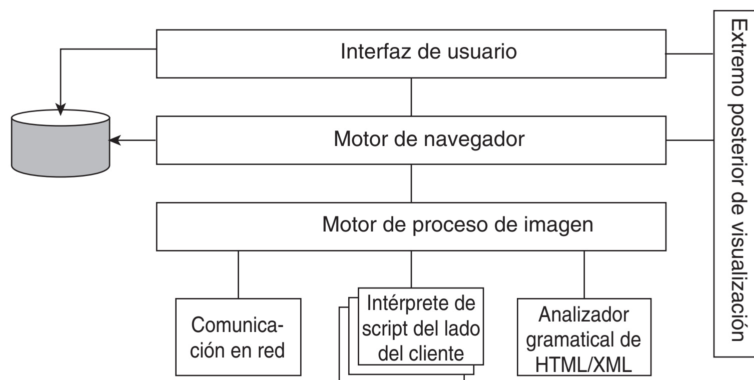


Figura 12-5. Componentes lógicos de un navegador web.

Un aspecto importante de los navegadores web es que deberán ser (idealmente) independientes de la plataforma. Este objetivo a menudo se logra por medio de bibliotecas de gráficas estándar, mostradas como el respaldo de lo visualizado en pantalla, junto con bibliotecas estándar de red.

El núcleo del navegador está formado por un motor de navegación y el motor de proceso de imagen. Este segundo elemento contiene todo el código necesario para mostrar apropiadamente los documentos en pantalla como ya explicamos. Este proceso de imagen requiere, por lo menos, el análisis sintáctico del HTML o del XML, aunque también puede requerir interpretación de script. En la mayoría de los casos, se incluye sólo un intérprete de Javascript, aunque en teoría también pueden estar incluidos otros intérpretes. El motor del navegador proporciona los mecanismos indispensables para que el usuario final revise el documento, seleccione algunas partes, active hipervínculos, etcétera.

Uno de los problemas que los diseñadores del navegador web tienen que enfrentar es que deberá ser fácil de ampliar de modo que, en principio, pueda soportar cualquier tipo de documento que le sea regresado por un servidor. El método seguido en la mayoría de los casos es ofrecer medios conocidos como plug-ins. Como ya mencionamos, un **plug-in** es un programa pequeño que puede ser dinámicamente cargado en un navegador para manejar un tipo de documento específico. El plug-in se da como tipo MIME. Un plug-in deberá estar disponible localmente, quizá después de ser específicamente transferido por un usuario desde un servidor remoto. Los plug-in normalmente ofrecen una interfaz estándar al navegador y, de igual modo, esperan una interfaz estándar del navegador. Lógicamente, forman una extensión del motor de proceso de imagen mostrado en la figura 12-5.

Otro proceso del lado del cliente que a menudo se utiliza es el **proxy web** (Luotonen y Altis, 1994). Originalmente, se utilizaba un proceso como ese para permitir que un navegador manejara protocolos a nivel de aplicación diferentes de HTTP, como se muestra en la figura 12-6. Por ejemplo, para transferir un archivo desde un servidor FTP, el navegador puede emitir una solicitud HTTP a un proxy FTP local, el cual buscará el archivo y lo devolverá embebidos como HTTP.

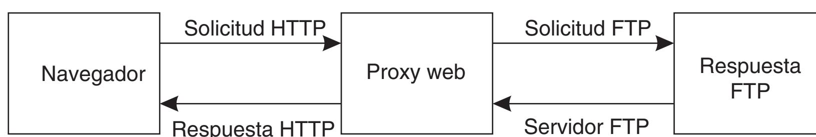


Figura 12-6. Utilización de un proxy web cuando el navegador no habla FTP.

Por el momento, la mayoría de los navegadores son capaces de soportar varios protocolos, o de lo contrario pueden ser ampliados dinámicamente para que lo hagan, y por esa razón no se requiere proxies. Sin embargo, éstos siguen siendo utilizados por otras razones. Por ejemplo, un proxy puede ser configurado para que filtre solicitudes y respuestas (acercándolo al cortafuego a nivel de aplicación), registro, compresión, pero sobre todo para almacenamiento en caché. Más adelante regresamos al tema de almacenamiento en caché del proxy. Un proxy Web ampliamente utilizado es **Squid**, el cual ha sido desarrollado como un proyecto de fuente abierta. Información detallada sobre Squid se encuentra en Wessels (2004).

12.2.2 Servidor web Apache

Por mucho, el servidor web más popular es Apache, que se estima es utilizado para alojar aproximadamente el 70% de todos los sitios web. Apache es una compleja pieza de software, y con las numerosas mejoras a los tipos de documentos que ahora son ofrecidos por la web, es importante que el servidor sea altamente configurable y extensible, y al mismo tiempo muy independiente de plataformas específicas.

Implementar al servidor independiente de la plataforma se lleva a cabo, en esencia, proporcionando su propio entorno en tiempo de ejecución básico, el que luego es implementado para diferentes sistemas operativos. Este entorno en tiempo de ejecución, conocido como **Apache portátil en tiempo de ejecución** (**APR**, por sus siglas en inglés), es una biblioteca que proporciona una interfaz independiente de la plataforma para manejo de archivos, interconexión en redes, bloqueo, hilos, y así sucesivamente. Cuando se amplía Apache (como se verá en breve) la portatibilidad se garantiza en gran medida siempre que sólo se hagan llamadas al APR y se eviten las llamadas a bibliotecas propias de una plataforma.

Como se mencionó, Apache fue diseñado no sólo para que fuera flexible (en el sentido de que puede ser configurado para incluir considerables detalles) sino también para que fuera relativamente fácil ampliar su funcionalidad. Por ejemplo, más adelante en este capítulo estudiaremos una replicación adaptable en Globule, una red de entrega de contenido elaborado en casa y desarrollada en la Universidad Vrije de Amsterdam. Globule se implementó como una extensión de Apache, basado en el APR, pero también muy independiente de otras extensiones desarrolladas para Apache.

Desde cierta perspectiva, Apache puede ser considerado como un servidor completamente general diseñado para producir una respuesta a una solicitud entrante. Desde luego, existen muchas clases de dependencias y suposiciones ocultas por las cuales Apache resulta ser principalmente

adecuado para manejar solicitudes de documentos web. Por ejemplo, como ya mencionamos, los navegadores y servidores web utilizan HTTP como su protocolo de comunicación. El HTTP virtualmente siempre se implementa encima del TCP, razón por la cual el núcleo de Apache asume que todas las solicitudes entrantes se adhieren a una forma de comunicación orientada a una conexión basada en TCP. Las solicitudes basadas en, por ejemplo, UDP no pueden ser manejadas apropiadamente sin que se modifique el núcleo Apache.

Sin embargo, el núcleo Apache hace pocas suposiciones sobre cómo deberán ser manejadas las solicitudes entrantes. Su organización total se muestra en la figura 12-7. Para esta organización resulta fundamental el concepto de **gancho**, el cual no es nada más que un guardalugar para un grupo específico de funciones. El núcleo Apache asume que las solicitudes se procesan en varias fases compuestas cada una por algunos ganchos. Cada gancho representa, por tanto, un grupo similar de acciones que deben ser ejecutadas como parte del procesamiento de una solicitud.

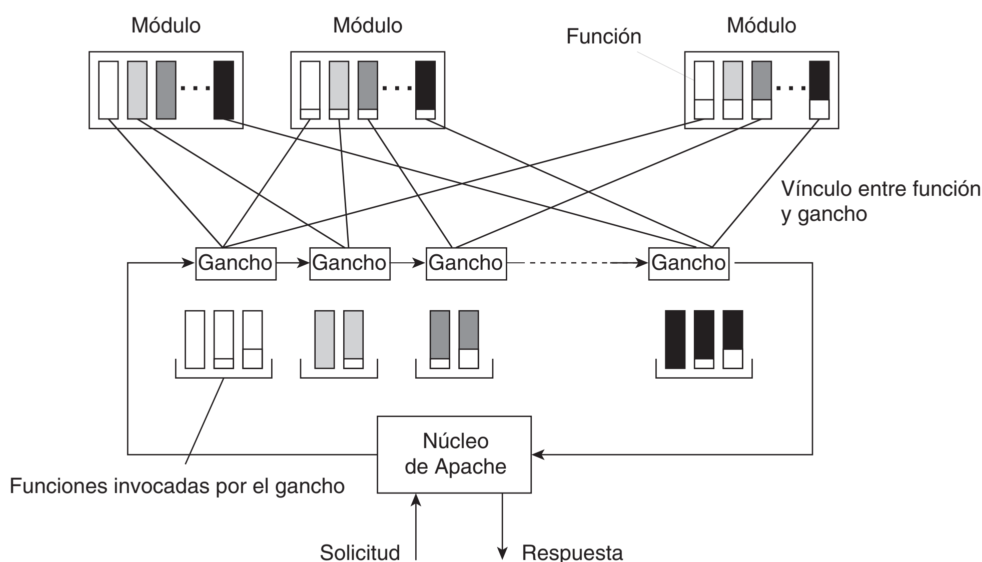


Figura 12-7. Organización general del servidor web Apache.

Por ejemplo, existe un gancho para traducir una URL a un nombre de archivo local. Tal traducción casi siempre tiene que ser realizada cuando se procesa una solicitud. De igual modo, existe un gancho para escribir información en un registro, un gancho para verificar la identificación de un cliente, un gancho para verificar los derechos de acceso, y un gancho para comprobar con qué tipo de MIME está relacionada la solicitud (por ejemplo, para asegurarse de que una solicitud puede ser manejada apropiadamente). Como se muestra en la figura 12-7, los ganchos se procesan en un orden predeterminado. Es aquí donde observamos explícitamente que Apache aplica un flujo de control específico por lo que se refiere al procesamiento de solicitudes.

Todas las funciones asociadas con un gancho son provistas por **módulos** distintos. Aunque en principio un desarrollador podría cambiar el conjunto de ganchos que serán procesados por Apache,

es mucho más común escribir módulos que contengan las funciones a ser llamadas como parte del procesamiento de ganchos estándar por el servidor Apache no modificado. El principio subyacente es bastante simple. Cada gancho puede contener un conjunto de funciones donde cada función deberá corresponder a un prototipo específico (es decir, una lista de parámetros y tipo de retorno). Un desarrollador de módulos escribirá funciones para ganchos específicos. Cuando compila Apache, el desarrollador especifica qué función deberá ser agregada a cada gancho. Esto se muestra en la figura 12-7 como los diversos vínculos entre funciones y ganchos.

Como puede haber decenas de módulos, generalmente cada gancho contendrá varias funciones. Es normal que los módulos se consideren mutuamente independientes, de modo que las funciones implementadas en el mismo gancho serán ejecutadas en algún orden arbitrario. Sin embargo, Apache también puede manejar dependencias de módulos al permitir que un desarrollador especifique el orden en que las funciones de diferentes módulos deberán ser procesadas. En general, el resultado es un servidor web extremadamente versátil. Información detallada sobre cómo configurar Apache, y una buena introducción a cómo puede ser ampliado se encuentran en Laurie y Laurie (2002).

12.2.3 Servidores web basados en clusteres

Un importante problema relacionado con la naturaleza cliente-servidor de la web es que un servidor puede sobrecargarse con facilidad. Una solución práctica empleada en muchos diseños es simplemente replicar un servidor en un cluster de servidores y utilizar un mecanismo distinto, tal como un extremo frontal, para direccionar las solicitudes de los clientes hacia una de las réplicas. Este principio se muestra en la figura 12-8 y es un ejemplo de distribución horizontal como lo analizamos en el capítulo 2.

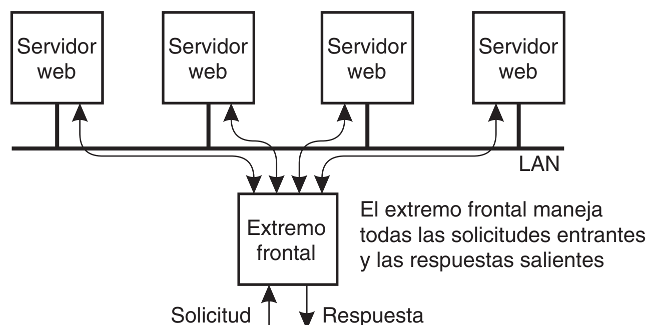


Figura 12-8. Principio de la utilización de un cluster de servidores en combinación con un extremo frontal para implementar un servicio web.

Un aspecto crucial de esta organización es el diseño del frente de despacho, ya que se puede convertir en un serio cuello de botella para el desempeño por todo el tráfico que pasará por ahí. En general, se hace una distinción entre el frente de despacho que operan como conmutadores de capa de transporte y aquellos que operan al nivel de capa de aplicación.

Siempre que un cliente emite una solicitud HTTP, establece una conexión TCP con el servidor. Un conmutador de capa de transporte simplemente pasa los datos enviados a lo largo de la conexión TCP a uno de los servidores, de acuerdo con alguna medición de la carga del servidor. La respuesta de dicho servidor es regresada al conmutador, el cual entonces la remitirá al cliente solicitante. Como una optimización, el conmutador y los servidores pueden colaborar en la implementación de un **TCP handoff**, como vimos en el capítulo 3. La desventaja principal de un conmutador de capa de transporte es que el conmutador no puede tomar en cuenta el contenido de la solicitud HTTP enviada a lo largo de la conexión TCP. Cuando mucho, sólo puede basar sus decisiones de redirección en las cargas del servidor.

Como regla general, un método mejor es utilizar una **solicitud de distribución consciente del contenido**, mediante la cual el frente de despacho primero inspecciona una solicitud HTTP entrante y luego decide a qué servidor deberá ser remitida. La distribución consciente del contenido ofrece varias ventajas. Por ejemplo, si el frente de despacho siempre remite solicitudes para el mismo documento al mismo servidor, dicho servidor puede ser capaz de efectivamente guardar en la memoria caché el documento con el resultado de tiempos de respuesta más rápidos. Además, es posible distribuir realmente el conjunto de documentos entre los servidores en lugar de tener que replicar cada documento para cada servidor. Este método utiliza con más eficiencia la capacidad de almacenamiento disponible y permite utilizar servidores dedicados para manejar documentos especiales tales como los de audio y video.

Un problema con la distribución consciente del contenido es que el extremo frontal tiene que realizar mucho trabajo. Idealmente, desearíamos tener la eficiencia del TCP handoff y la funcionalidad de la distribución consciente del contenido. Lo que se debe hacer es distribuir el trabajo del frente de despacho y combinar con el conmutador de capa de transporte, como se propone en Aron y colaboradores (2000). En combinación con el TCP handoff, el extremo frontal realiza dos tareas. Primero, cuando inicialmente llega una solicitud, debe decidir qué servidor manejará el resto de la comunicación con el cliente. En segundo lugar, el extremo frontal deberá remitir los mensajes TCP del cliente asociados con la conexión TCP transferida.

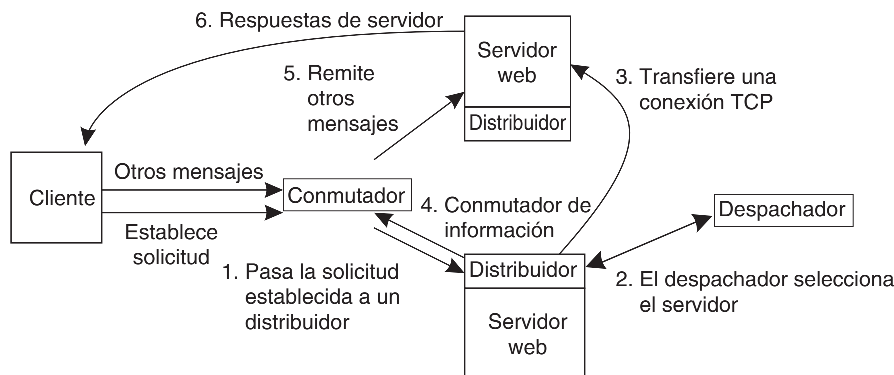


Figura 12-9. Cluster escalable de servidores web conscientes del contenido.

Estas dos tareas pueden ser distribuidas como se muestra en la figura 12-9. El despachador es responsable de decidir a qué servidor deberá ser transferida una conexión TCP; un distribuidor monitorea el tráfico TCP entrante para una conexión transferida. El conmutador se utiliza para enviar mensajes TCP a un distribuidor. Cuando un cliente se pone en contacto por primera vez con el servicio web, su TCP de establecimiento de conexión es remitido a un distribuidor, el que a su vez se pone en contacto con el despachador para decidir a qué servidor deberá ser transferida la conexión. En ese momento, el conmutador recibe una notificación de que deberá enviar todos los mensajes TCP a través de esa conexión al servidor seleccionado.

Existen otras alternativas distintas y más refinamientos para establecer grupos de servidores web. Por ejemplo, en lugar de cualquier clase de frente de despacho, también es posible utilizar un **round robin DNS** mediante el cual un solo nombre de dominio se asocia con múltiples direcciones IP. En este caso, cuando se resuelve el nombre del servidor de un sitio web, un navegador de cliente recibiría una lista de múltiples direcciones, correspondiendo cada dirección a uno de los servidores web. Normalmente, los navegadores eligen la primera dirección de la lista. Sin embargo, lo que hace un servidor DNS popular tal como **BIND** es poner en circulación la lista de entradas que regresa (Albitz y Liu, 2001). Por consiguiente, se obtiene una distribución simple de solicitudes a través de los servidores que conforman el cluster.

Por último, también es posible no utilizar cualquier clase de intermediario sino simplemente dar a cada servidor web la misma dirección IP. En ese caso, se debe asumir que todos los servidores están conectados a través de una única LAN de transmisión. Lo que sucederá es que al llegar una solicitud HTTP, el enrutador IP conectado a la LAN simplemente la remite a todos los servidores, los que luego ejecutarán el mismo algoritmo distribuido para decidir determinísticamente cuál de ellos manejará la solicitud.

En un excelente estudio de Cardellini y colaboradores (2002) se describen las diferentes formas de organizar clusteres en web y alternativas como las que hemos analizado aquí. El lector interesado deberá consultar dicho artículo para obtener más detalles y referencias.

12.3 COMUNICACIÓN

Cuando se trata de sistemas distribuidos basados en Web, se utilizan sólo algunos protocolos de comunicación. En primer lugar, para sistemas web tradicionales, HTTP es el protocolo estándar para intercambiar mensajes. En segundo lugar, cuando se consideran servicios web, SOAP es la forma preestablecida de intercambiar mensajes. Ambos protocolos serán abordados con cierto detalle en esta sección.

12.3.1 Protocolo de transferencia de hipertexto

En la web, toda la comunicación entre clientes y servidores está basada en el **protocolo de transferencia de hipertexto (HTTP**, por sus siglas en inglés). El HTTP es un protocolo cliente-servidor relativamente simple: un cliente envía un mensaje de solicitud a un servidor y espera un mensaje

de respuesta. Una propiedad importante del HTTP es que es un protocolo sin estado. En otros términos, no tiene cualquier concepto de conexión abierta y no requiere que un servidor mantenga información sobre sus clientes. El HTTP se describe en Fielding y colaboradores (1999).

Conexiones HTTP

El HTTP está basado en TCP. Siempre que un cliente envía una solicitud a un servidor, primero establece una conexión TCP con el servidor y luego envía su mensaje de solicitud a través de la conexión. Se utiliza la misma conexión para recibir la respuesta. Utilizando TCP como su protocolo subyacente, HTTP no tiene que preocuparse por solicitudes y respuestas perdidas. Un cliente y un servidor simplemente asumen que sus mensajes sí llegaron al otro lado. Si las cosas resultaran mal, por ejemplo, cuando la conexión se corta o ocurre un tiempo fuera, se reporta un error. Sin embargo, en general, no intenta recuperarse de la falla.

Uno de los problemas con las primeras versiones de HTTP fue su uso ineficiente de las conexiones TCP. Cada documento web se construía con un conjunto de diferentes archivos tomados del mismo servidor. Para desplegar apropiadamente un documento, es necesario que esos archivos también se transfieran al cliente. Cada uno de estos archivos en, en principio, es simplemente otro documento para el cual el cliente puede emitir una solicitud distinta al servidor donde están guardados.

En las versiones 1.0 y anteriores de HTTP, cada solicitud a un servidor requería establecer una conexión aparte, como se muestra en la figura 12-10(a). Cuando el servidor respondía, la conexión se interrumpía de nuevo. Esas conexiones se conocen como **no persistentes**. Una desventaja importante de las conexiones no persistentes es que resulta relativamente costoso establecer una conexión TCP. Por consiguiente, el tiempo requerido para transferir un documento completo junto con todos sus elementos a un cliente puede ser considerable.

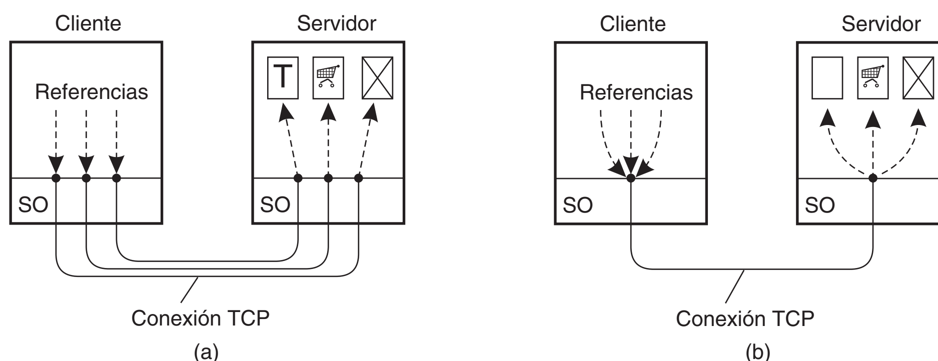


Figura 12-10. (a) Utilización de conexiones no persistentes. (b) Utilización de conexiones persistentes.

Observemos que el HTTP no impide que un cliente establezca varias comunicaciones al mismo tiempo con el mismo servidor. A menudo se utiliza este método para ocultar la latencia provocada

por el tiempo requerido para establecer la conexión y transferir datos en paralelo del servidor al cliente. Muchos navegadores utilizan este método para mejorar el desempeño.

Otro método, seguido en la versión 1.1 de HTTP, es utilizar una **conexión persistente**, con la cual se pueden enviar varias solicitudes (y sus respectivas respuestas), sin la necesidad de una conexión distinta para cada par de (*solicitud-respuesta*). Para mejorar aún más el desempeño, un cliente puede emitir varias solicitudes en fila sin esperar la respuesta a la primera solicitud (proceso también conocido como **canalización** o **pipelining**). El uso de conexiones persistentes se ilustra en la figura 12-10(b).

Métodos HTTP

El HTTP ha sido designado como protocolo cliente-servidor de uso general orientado a la transferencia de documentos en ambas direcciones. Un cliente puede solicitar que cada una de estas operaciones sea realizada en el servidor enviando un mensaje de solicitud que contenga la operación deseada al servidor. En la figura 12-11 proporcionamos una lista de los mensajes de solicitud más comúnmente utilizados.

Operación	Descripción
Head	Solicitud de regresar el encabezado de un documento
Get	Solicitud de regresar un documento al cliente
Put	Solicitud de guardar un documento
Post	Proporcionar datos que se van a agregar a un documento (conjunto)
Delete	Solicitud de eliminar un documento

Figura 12-11. Operaciones soportadas por HTTP.

El HTTP asume que cada documento puede tener metadatos asociados, los cuales están guardados en un encabezado aparte enviado junto con una solicitud o respuesta. La operación head es entregada al servidor cuando un cliente no desea el documento en sí, sino sólo sus metadatos asociados. Por ejemplo, el uso de la operación head regresará la hora en que el documento referido fue modificado. Se puede utilizar esta operación para verificar la validez del documento guardado en caché por el cliente. También puede ser utilizada para verificar si existe un documento, sin tener que transmitir en realidad el documento.

La operación más importante es get. Esta operación se utiliza para obtener en realidad un documento del servidor y regresarlo al cliente solicitante. Además, es posible especificar que un documento deberá ser regresado sólo si se modificó después de cierto tiempo. También, el HTTP permite que los documentos tengan **etiquetas** asociadas (cadenas de caracteres) y obtener un documento sólo si concuerda con ciertas etiquetas.

La operación put es la opuesta de get. Un cliente puede solicitar que un servidor guarde un documento bajo un nombre dado (el cual se envía junto con la solicitud). Desde luego, generalmente un servidor no ejecutará operaciones put a ciegas, sino que sólo aceptará solicitudes de clientes autorizados. Cómo tratar con estos temas de seguridad lo veremos más adelante.

La operación post es algo similar a guardar un documento, excepto que un cliente solicitará que se agreguen datos a un documento o a un conjunto de documentos. Un ejemplo típico es fijar un artículo en un grupo de noticias. La característica distintiva, comparada con la operación put, es que una operación post indica a qué grupo de documentos deberá ser "agregado" un artículo. El artículo se envía junto con la solicitud. Por contraste, la operación put porta un documento y el nombre bajo el cual se le pidió guardarlo.

Por último, la operación delete se utiliza para solicitar a un servidor que elimine el documento nombrado en el mensaje enviado al servidor. De nueva cuenta, si en realidad ocurre o no la eliminación depende de varias medidas de seguridad. Incluso puede ser que el propio servidor no tenga los permisos adecuados para eliminar el documento referido. Después de todo, el servidor es simplemente un proceso de usuario.

Mensajes HTTP

Toda la comunicación entre un cliente y un servidor ocurre a través de mensajes. El HTTP sólo reconoce mensajes de solicitud y respuesta. Un mensaje de solicitud se compone de tres partes, como se muestra en la figura 12-12(a). La **línea de solicitud** es obligatoria e identifica la operación que el cliente desea que el servidor realice junto con una referencia al documento asociado con la solicitud. Se utiliza otro campo para identificar la versión del HTTP que el cliente está esperando. A continuación explicamos los encabezados de mensaje adicionales.

Un mensaje de respuesta inicia con la **línea de solicitud** que contiene el número de versión y un código de estado de tres dígitos, como se muestra en la figura 12-12(b). El código se explica brevemente con una frase textual enviada como parte de la línea de estado. Por ejemplo, el código de estado 200 indica que una solicitud podría ser aceptada, y tiene la frase asociada "OK". Otros códigos de uso frecuente son:

400 (Solicitud errónea)

403 (Prohibida)

404 (No encontrada)

Un mensaje de solicitud o respuesta puede contener encabezados adicionales. Por ejemplo, si un cliente solicitó la operación post para un documento de sólo lectura, el servidor responderá con un mensaje que tenga el código 405 ("Método no permitido") junto con el encabezado de mensaje *Allow* (*Permitir*) que especifica las operaciones permitidas (por ejemplo, head y get). Como otro ejemplo, un cliente puede estar interesado sólo en un documento si no ha sido modificado desde cierto tiempo *T*. En ese caso, la solicitud get del cliente se aumenta con el encabezado de mensaje *IF-Modified-Since* que especifica el valor *T*.

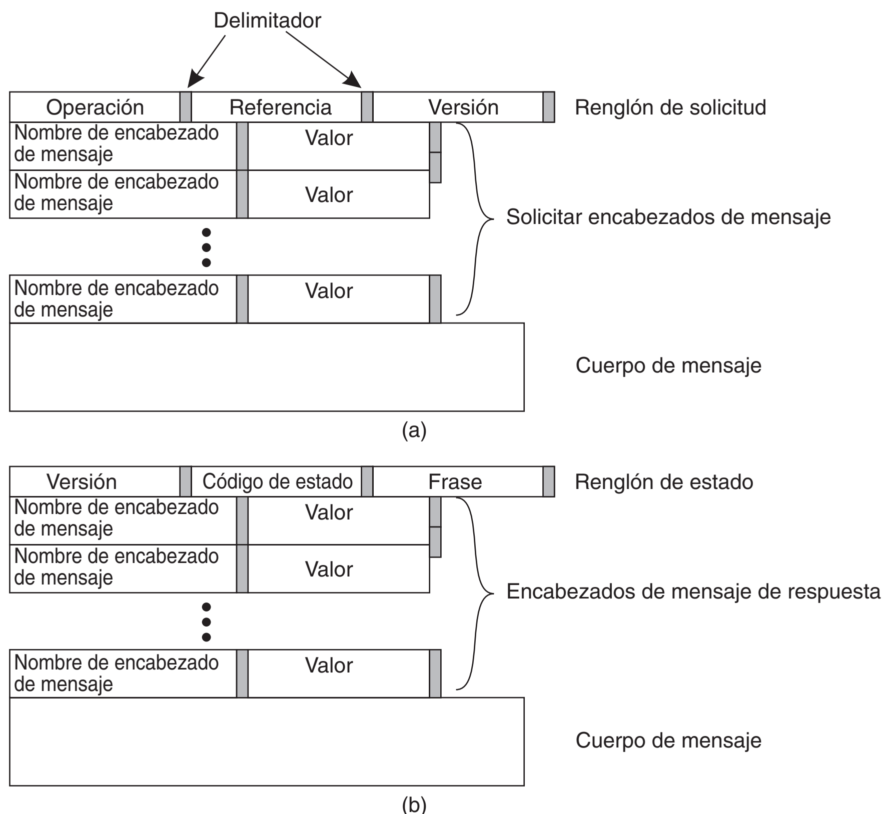


Figura 12-12. (a) Mensaje de solicitud HTTP. (b) Mensaje de respuesta HTTP.

La figura 12-13 muestra varios encabezados de mensaje válidos que pueden ser enviados junto con una solicitud o respuesta. La mayoría de los encabezados son autoexplicativos, por lo que no los presentaremos en forma individual.

Existen varios encabezados de mensaje que el cliente puede enviar al servidor explicando lo que puede aceptar como respuesta. Por ejemplo, un cliente puede ser capaz de aceptar respuestas que han sido comprimidas con el programa de compresión *gzip* disponible en la mayoría de las plataformas Windows y UNIX. En ese caso, el cliente enviará un encabezado de mensaje *Accept-Encoding* junto con su solicitud, con su contenido que incluye "Accept-Encoding:gzip". Asimismo, podemos utilizar un encabezado de mensaje *Accept* para especificar, por ejemplo, que sólo pueden ser regresadas páginas web HTML.

Existen dos encabezados de mensaje de seguridad, pero como veremos más adelante en esta sección, en la web la seguridad casi siempre es manejada por un protocolo de capa de transporte distinto.

Los encabezados de mensaje *Location* y *Referer* se utilizan para **redireccionar** un cliente hacia otro documento (observemos que "Referer" está mal escrito en la especificación). Redireccionar corresponde al uso de apuntadores de remisión para localizar un documento, como explicamos

Encabezado	Origen	Contenido
Aceptar	Cliente	Tipo de documentos que el cliente puede manejar
Aceptar conjunto de caracteres	Cliente	Conjuntos de caracteres aceptables para el cliente
Aceptar codificación	Cliente	Codificaciones del documento que el cliente puede manejar
Aceptar lenguaje	Cliente	Lenguaje natural que el cliente puede manejar
Autorización	Cliente	Lista de las credenciales del cliente
Autentificar WWW	Servidor	Desafío de seguridad al que el cliente deberá responder
Fecha	Ambos	Fecha y hora en que el mensaje fue enviado
Etiqueta E	Servidor	Etiquetas asociadas con el documento regresado
Expira	Servidor	Tiempo durante el cual la respuesta permanece válida
De	Cliente	Dirección de correo electrónico del cliente
Servidor	Cliente	Nombre DNS del servidor del documento
Si concuerda	Cliente	Etiquetas que el documento debe tener
Si ninguno concuerda	Cliente	Etiquetas que el documento no deberá tener
Si se modificó desde	Cliente	Le pide al servidor que regrese un documento sólo si ha sido modificado desde el tiempo especificado
Si no se modificó desde	Cliente	Le pide al servidor que regrese un documento sólo si no ha sido modificado desde el tiempo especificado
Última vez que se modificó	Servidor	Tiempo en que el documento regresado fue modificado por última vez
Ubicación	Servidor	Referencia a un documento al cual el cliente deberá redireccionar su solicitud
Referente	Cliente	Se refiere al documento más recientemente solicitado del cliente
Actualización	Ambos	Protocolo de aplicación al cual desea cambiarse al remitente
Advertencia	Ambos	Información sobre el estado de los datos en el mensaje

Figura 12-13. Algunos encabezados de mensajes HTTP.

en el capítulo 5. Cuando un cliente emite una solicitud para un documento D, tal vez el servidor responda con un encabezado de mensaje *Location*, el cual especifica que el cliente deberá reemitir la solicitud, aunque ahora para el documento D'. Cuando utiliza la referencia a D', el cliente puede agregar un encabezado de mensaje *Referer* que contenga la referencia a D para indicar qué provocó el redireccionamiento. En general, este encabezado de mensaje se utiliza para indicar el documento más recientemente solicitado por el cliente.

El encabezado de mensaje *Upgrade* se utiliza para cambiar a otro protocolo. Por ejemplo, el cliente y el servidor pueden utilizar HTTP/1.1 inicialmente sólo para tener una forma genérica de establecer una conexión. El servidor puede responder de inmediato informando al cliente que desea continuar la comunicación con una versión segura de HTTP, tal como SHTTP (Rescorla y Schiffman, 1999). En ese caso, el servidor enviará un encabezado de mensaje *Upgrade* con el contenido "Upgrade:SHTTP".

12.3.2 Protocolo de acceso a un objeto simple

Donde HTTP es el protocolo de comunicación estándar para sistemas distribuidos basados en la web tradicionales, el **protocolo de acceso a un objeto simple** (**SOAP**, por sus siglas en inglés) conforma el estándar de comunicación con servicios web (Gudgin y cols., 2003). SOAP ha vuelto más importante al HTTP de lo que ya era: la mayoría de las comunicaciones se implementan mediante HTTP. SOAP, por sí mismo, no es un protocolo difícil. Su propósito principal es proporcionar medios relativamente simples para permitir que diferentes partes que pueden saber muy poco una de otra sean capaces de comunicarse. En otros términos, el protocolo está diseñado con la suposición de que dos partes que se están comunicando tienen muy poco conocimiento común.

Con base en esta suposición, no sorprende que los mensajes SOAP estén basados en gran medida en XML. Recordemos que XML es un lenguaje de metamarcado, ello significa que una descripción XML incluye la definición de los elementos utilizados para describir un documento. En la práctica, esto significa que la definición de la sintaxis tal como se utiliza para un mensaje es una parte de éste. El uso de esta sintaxis permite que un receptor realice el análisis gramatical de tipos muy diferentes de mensajes. Desde luego, el *significado* de un mensaje sigue siendo indefinido, y por tanto también las acciones que se deben tomar cuando llega un mensaje. Si el receptor no le encuentra sentido a su contenido, no puede haber progreso.

Un mensaje SOAP, en general, se compone de dos partes, las cuales insertan conjuntamente lo que se conoce como **envolvente SOAP**. El cuerpo contiene el mensaje en sí, en tanto que el encabezado es opcional ya que contiene información pertinente para los nodos ubicados a lo largo de la ruta que va desde el remitente hasta el destinatario. Típicamente, estos nodos se componen de los diversos procesos aplicados en una implementación de múltiples niveles de un servicio web. Todo en la envolvente está expresado en XML, es decir, el encabezado y el cuerpo.

Por extraño que parezca, una envolvente SOAP no contiene la dirección del destinatario. En vez de eso, SOAP asume explícitamente que el protocolo utilizado para transferir los mensajes especifica el destinatario. Con esta finalidad, SOAP especifica **enlaces** a protocolos de transferencia subyacentes. Por el momento, existen dos enlaces: una a HTTP y otra a SMTP, el protocolo de transferencia de correo en internet. Así que, por ejemplo, cuando un mensaje SOAP está ligado a un HTTP, el destinatario será especificado en la forma de un URL, en tanto que un enlace a SMTP especificará el destinatario en la forma de una dirección de correo electrónico.

Estos dos tipos diferentes de enlaces también indican dos estilos diferentes de interacciones. La primera, más común, es el **estilo de intercambio conversacional**. En este estilo, esencialmente, dos partes intercambian documentos estructurados. Por ejemplo, un documento de esta clase puede contener una orden de compra completa como la que se llena al reservar electrónicamente un vuelo. La respuesta a una orden como esa podría ser un documento de confirmación, que contenía entonces el número de orden, información sobre el vuelo, un asiento reservado, y quizá también un código de barras que deba ser escaneado al abordar el avión.

Por contraste, un **intercambio estilo RPC** se apega más al comportamiento de respuesta a una solicitud tradicional cuando se invoca un servicio web. En este caso, el mensaje SOAP identificará explícitamente el procedimiento a ser invocado, y también proporcionará una lista de valores de

parámetro como entrada para dicha invocación. Asimismo, la respuesta será un mensaje formal que contiene la respuesta a la invocación.

Por lo general, un intercambio estilo RPC es soportado mediante un enlace a http, en tanto que el mensaje estilo conversacional se vinculará o a SMTP o a HTTP. Sin embargo, en la práctica, la mayoría de los mensajes SOAP se envían a través de HTTP.

Una importante observación es que, aunque el XML es mucho más fácil de usar que un analizador gramatical porque las definiciones sintácticas forman entonces parte del mensaje, la propia sintaxis XML es extremadamente detallada. Como una consecuencia, en la práctica, el análisis gramatical de mensajes XML a menudo introduce un serio cuello de botella que afecta el desempeño (Allman, 2003). A este respecto, es un tanto sorprendente que la mejoría al desempeño del XML reciba relativamente poca atención, aunque las soluciones están en proceso (vea, por ejemplo, Kostoulas y cols., 2006).

```
<env:Envelope xmlns: env="http://www.w3.org/2003/05/soap-envelope">
    <env:Header>
        <n:alertcontrol xmlns:n="http://example.org/alertcontrol">
            <n:priority>1</n:priority>
            <n:expires>2001-06-22T14:00:00-05:00</n:expires>
        </n:alertcontrol>
    </env:Header>
    <env:Body>
        <m:alert xmlns:m="http://example.org/alert">
            <m:msg>Pick up Mary at school at 2pm</m:msg>
        </m:alert>
    </env:Body>
</env:Envelope>
```

Figura 12-14. Ejemplo de un mensaje SOAP basado en XML.

Lo que también sorprende es que muchas personas crean que las especificaciones XML pueden ser convenientemente leídas por seres humanos. El ejemplo mostrado en la figura 12-14 se tomó de la especificación SOAP oficial (Gudgin y cols., 2003). Descubrir lo que este mensaje SOAP transmite requiere cierta dosis de investigación, y no es difícil imaginar que la oscuridad puede aparecer como un subproducto natural del uso de XML. La pregunta que viene entonces a la mente, acerca de si el método basado en texto tal como lo sigue el XML ha sido el correcto, se responde así: nadie podría leer de manera conveniente documentos XML, y la velocidad de los analizadores gramaticales se reduciría en forma severa.

12.4 ASIGNACIÓN DE NOMBRES

La web utiliza un solo sistema de asignación de nombres para referirse a documentos. Los nombres utilizados se llaman **identificadores de recursos uniformes** o simplemente **URI**, por sus siglas en

inglés (Berners-Lee y cols., 2005). Los URIs se presentan en dos formas. Un l**ocalizador de recursos uniforme** (**URL**) es un URI que identifica un documento e incluye información sobre cómo y dónde accesarlo. En otros términos, un URL es una referencia a un documento que depende de la localización. Por contraste, un **nombre de recursos uniforme** (**URN**, por sus siglas en inglés) actúa como identificador verdadero tal como vimos en el capítulo 5. Un URN se utiliza como referencia globalmente única a un documento, independiente de la localización, y persistente.

La sintaxis de un URI está determinada por su **esquema** asociado. El nombre de un esquema es una parte del URI. Se han definido muchos esquemas diferentes, y a continuación mencionaremos algunos junto con ejemplos de sus URI asociados. El esquema *http* es el más conocido, pero no el único. También debemos observar que la diferencia entre un URL y un URN se reduce gradualmente. En cambio, ahora es común definir simplemente espacios de nombre URI [vea también Daigle y cols. (2002)].

En el caso de URLs, a menudo vemos que contienen información sobre cómo y dónde acceder a un documento. Cómo acceder a un documento, en general, es reflejado por el nombre del esquema que forma parte del URL, tal como *http*, *ftp*, o *telnet*. Dónde está localizado un documento se encuentra insertado en un URL por medio del nombre DNS del servidor al cual se puede enviar una solicitud de acceso, aunque también es posible utilizar una dirección IP. El número de puerto por donde el servidor escuchará las solicitudes también forma parte del URL; cuando se deja fuera, se usa un puerto preestablecido. Por último, un URL también contiene el nombre del documento que buscará el servidor, lo cual conduce a las estructuras generales mostradas en la figura 12-15.

Esquema	Nombre del servidor	Nombre de ruta
http ://	www.cs.vu.nl	/home/steen/mbox

(a)

Esquema	Nombre del servidor	Puerto	Nombre de ruta
http ://	www.cs.vu.nl	: 80	/home/steen/mbox

(b)

Esquema	Nombre del servidor	Puerto	Nombre de ruta
http ://	130.37.24.11	: 80	/home/steen/mbox

(c)

Figura 12-15. Estructuras utilizadas a menudo para URLs. (a) Utilizando sólo un nombre DNS. (b) Combinando un nombre DNS con un número de puerto. (c) Combinando una dirección IP.

La resolución de un URL tal como los mostrados en la figura 12-15 es simple. Si se hace referencia al servidor mediante su nombre DNS, dicho nombre tendrá que ser resuelto para la dirección IP del servidor. Utilizando el número de puerto contenido en el URL, el cliente puede entonces ponerse en contacto con el servidor usando el protocolo nombrado por el esquema, y pasarle el nombre del documento que forma la última parte del URL.

Nombre	Utilizado para	Ejemplo
http	HTTP	http://www.cs.vu.nl:80/globe
mailto	Correo electrónico	mailto:steen@cs.vu.nl
ftp	FTP	ftp://ftp.cs.vu.nl/pub/minix/README
file	Archivo local	file:/edu/book/work/chp/11/11
data	Datos en línea	data:text/plain;charset=iso-8859-7,%e1%e2%e3
telnet	Inicio de sesión remota	telnet://flits.cs.vu.nl
tel	Teléfono	tel:+31201234567
modem	Módem	modem:+31201234567;type=v32

Figura 12-16. Ejemplos de URI.

Aunque los URL siguen siendo comunes en la web, se han propuesto varios espacios de nombre URI distintos para otras clases de recursos web. La figura 12-16 muestra varios ejemplos de URI. El URI *http* se utiliza para transferir documentos que usan HTTP como ya explicamos. Asimismo, existe un URI *ftp* para transferir archivos que utilizan FTP.

Una forma inmediata de documentos es soportada por URIs de *datos* (Masinter, 1998). En un URI de esta clase, el propio documento está incrustado en el URI, como cuando se empotran los datos de un archivo en un nodo índice (Mullender y Tanenbaum, 1984). El ejemplo de la figura 12-16 muestra un URI que contiene texto común para la cadena de letras griegas $\alpha\beta\gamma$.

Los URIs a menudo se utilizan también para otros propósitos, que hacen referencia a un documento. Por ejemplo, un URI *telnet* se usa para iniciar una sesión telnet con un servidor. También existen URIs para comunicación telefónica, tal como se describe en Schulzrinne (2005). El URI *tel*, en la forma que se muestra en la figura 12-16, en esencia se incrusta sólo un número telefónico y simplemente permite que el cliente establezca una llamada a través de la red telefónica. En este caso, el cliente típico será un teléfono. El URI *modem* puede ser utilizado para establecer una conexión basada en módem con otra computadora. En el ejemplo, figura 12-16, el URI establece que el módem remoto deberá adherirse al estándar ITU-T V32.

12.5 SINCRONIZACIÓN

La sincronización no ha sido tanto problema para la mayoría de los sistemas basados en la web por dos razones. Primero, la estricta organización de cliente-servidor de la web, donde los servidores nunca intercambian información con otros servidores (o clientes con otros clientes), significa que no hay mucho que sincronizar. En segundo lugar, la web puede ser considerada principalmente como un sistema de sólo lectura. Las actualizaciones en general son realizadas por una sola persona o entidad y difícilmente se presentan conflictos de escritura-escritura.

No obstante, las cosas están cambiando. Por ejemplo, existe una creciente demanda de proporcionar soporte a la autoría en colaboración de documentos web. En otros términos, la web deberá

soportar las actualizaciones concurrentes de documentos realizadas por usuarios o procesos en colaboración. Asimismo, con la introducción de servicios web, ahora se está viendo la necesidad de que los servidores se sincronicen entre sí y que sus acciones sean coordinadas. Ya estudiamos aquí la coordinación en servicios web. La sincronización del mantenimiento en colaboración de documentos web la abordaremos brevemente.

La autoría distribuida de documentos web es manejada por otro protocolo, conocido como **WebDAV** (Goland y cols., 1999). WebDAV significa **Autoría y control de versiones distribuidos en la web**, del inglés *Web Distributed Authoring and Versioning* y proporciona una forma simple de bloquear un documento compartido, y de crear, eliminar, copiar, y mover documentos desde servidores remotos. Enseguida describimos brevemente la sincronización tal como la soporta WebDAV. En Kim y colaboradores (2004) se encuentran los detalles sobre cómo utilizar WebDAV.

Para sincronizar el acceso concurrente a un documento compartido, WebDAV soporta un mecanismo de bloqueo simple. Existen dos tipos de bloqueo de escritura. Se puede asignar un bloqueo de escritura exclusivo a un solo cliente, y evitará que cualquier otro cliente modifique el documento compartido mientras esté bloqueado. También existe un bloqueo de escritura compartido, el cual permite que múltiples clientes actualicen simultáneamente el documento. Como el bloqueo ocurre en la granularidad de todo un documento, los bloqueos de escritura compartidos son convenientes cuando los clientes modifican diferentes partes del mismo documento. Sin embargo, los propios clientes no necesitan encargarse de que no ocurran conflictos de escritura-escritura.

La asignación de un bloqueo se realiza pasando un token de bloqueo al cliente solicitante. El servidor registra qué cliente tiene actualmente el token de bloqueo. Siempre que el cliente desea modificar el documento, envía una solicitud post HTTP al servidor, junto con el token de bloqueo. El token demuestra que el cliente tiene accceso de escritura al documento, por lo que el servidor realizará la solicitud.

Un tema importante de diseño es que no existe la necesidad de mantener conexiones entre el cliente y el servidor mientras persiste el bloqueo. El cliente simplemente puede desconectarse del servidor después de adquirir el bloqueo y reconectarse cuando envíe una solicitud HTTP.

Observe que un cliente se congela cuando mantiene un token de bloqueo, el servidor tendrá que reclamar el bloqueo de un modo u otro. WebDAV no especifica cómo deberán manejar los servidores éstas y otras situaciones similares, sino que lo deja abierto a implementaciones específicas. El razonamiento es que la mejor solución dependerá del tipo de documentos para el que se esté utilizando WebDAV. La razón de esto es que no existe una forma general de resolver en forma nítida el problema de bloqueos huérfanos.

12.6 CONSISTENCIA Y REPLICACIÓN

En sistemas distribuidos basados en la web, tal vez uno de los desarrollos más importantes es garantizar que el acceso a documentos web satisfaga estrictos requerimientos de desempeño y disponibilidad. Estos requerimientos condujeron a numerosas propuestas para guardar en caché y replicar contenido web, de las cuales analizaremos varias en esta sección. Donde los esquemas originales (que en gran

medida se siguen desplegando) han sido dirigidos hacia el soporte de contenido estático, también se está haciendo mucho esfuerzo para soportar contenido dinámico, es decir, documentos generados a consecuencia de una solicitud, así como también aquellos que contienen scripts. Una excelente y completa imagen de la replicación y el almacenamiento en caché de contenido web es provista por Rabinovich y Spatscheck (2002).

12.6.1 Almacenamiento en el caché de un proxy web

El almacenamiento en caché del lado del cliente ocurre generalmente en dos lugares. En el primer sitio, la mayoría de los navegadores están equipados con un sencillo medio de almacenamiento en caché. Siempre que un documento es buscado se guarda en el caché del navegador, desde donde se carga la siguiente vez. Los clientes, en general, pueden configurar en el almacenamiento en caché indicando en qué momento se deberá verificar la consistencia, tal como explicamos para el caso general a continuación.

En el segundo lugar, el sitio de un cliente con frecuencia ejecuta un proxy web. Como ya explicamos, un proxy web acepta solicitudes de clientes locales y las transfiere a servidores web. Cuando llega una respuesta, el resultado se transfiere al cliente. La ventaja de este método es que el proxy puede guardar en caché el resultado y regresarlo a otro cliente, si es necesario. En otros términos, un proxy web puede implementar un caché compartido.

Además de guardar en caché de navegadores y proxies, también es posible colocar cachés que cubran una región, o incluso un país, lo cual nos lleva a los **cachés jerárquicos**. Tales esquemas se utilizan principalmente para reducir el tráfico en la red, aunque tienen la desventaja de que potencialmente incurren en una latencia más alta en comparación con el uso de cachés no jerárquicos. Esta latencia más alta es provocada por la necesidad de que el cliente verifique múltiples cachés en lugar de sólo uno en el esquema no jerárquico. Sin embargo, esta latencia más alta está fuertemente relacionada con la popularidad de un documento: en el caso de documentos populares, la probabilidad de encontrar un copia en el caché más cercano al cliente es más alta que en el caso de un documento no popular.

Como una alternativa para la construcción de cachés jerárquicos, también se pueden organizar cachés de despliegue cooperativo como se muestra en la figura 12-17. En el **almacenamiento en caché cooperativo** o **almacenamiento en caché distribuido**, siempre que falla un caché en un proxy web, el proxy verifica primero varios proxies vecinos para ver si alguno contiene el documento solicitado. Si la verificación falla, el proxy remite la solicitud al servidor web responsable del documento. Este esquema se utiliza principalmente con cachés web que pertenecen a una misma organización o institución y están colocados en la misma LAN. Es interesante señalar que un estudio realizado por Wolman y colaboradores (1999) muestra que el almacenamiento en caché cooperativo puede ser efectivo solamente para grupos de clientes relativamente pequeños (del orden de decenas de miles de usuarios). Sin embargo, tales grupos también pueden ser servidos por medio de un solo caché de proxy, lo cual es mucho más barato en función de comunicación y uso de recursos.

Una comparación entre almacenamiento en caché jerárquico y caché cooperativo realizada por Rodriguez y colaboradores (2001) aclara que existen varios intercambios por hacer. Por ejemplo,

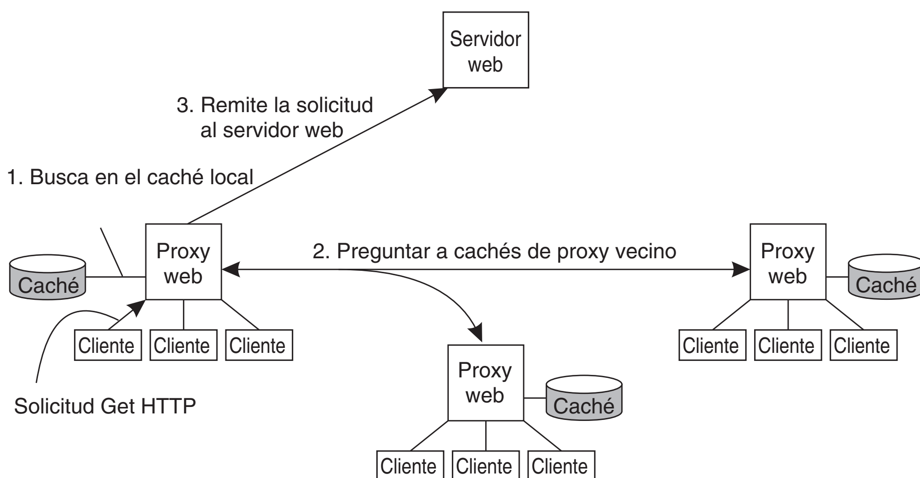


Figura 12-17. Principio de almacenamiento en caché cooperativo.

como los cachés cooperativos generalmente están conectados mediante enlaces de alta velocidad, el tiempo de transmisión requerido para buscar un documento es mucho más lento que para el caché jerárquico. También, como es de esperarse, los requerimientos de almacenamiento son menos estrictos para cachés cooperativos que para cachés jerárquicos. También, se encontró que las latencias esperadas son más bajas para cachés jerárquicos que para cachés distribuidos.

En la web se han desplegado diferentes protocolos de consistencia de caché. Para garantizar que un documento regresado por el caché sea consistente, algunos proxies web envían primero una solicitud HTTP get condicional al servidor con un encabezado de solicitud adicional *If-Modified-Since*, el cual especifica el tiempo de la última modificación asociado con el documento guardado en caché. Sólo si el documento cambió desde ese tiempo, el servidor lo regresará todo. De lo contrario, el proxy web puede regresar simplemente su versión guardada en caché al cliente local solicitante. Siguiendo la terminología introducida en el capítulo 7, esto corresponde a un protocolo basado en extracción.

Por desgracia, esta estrategia requiere que el proxy se ponga en contacto con un servidor por cada solicitud. Para mejorar el desempeño a expensas de consistencia más débil, el proxy web Squid (Wessels, 2004) asigna un tiempo de expiración $T_{expiración}$ que depende de cuándo fue modificado por última vez el documento al guardarlo en caché. En particular, si $T_{última_modificación}$ es el tiempo de modificación de un documento (registrado por su propietario), y $T_{guardado\ en\ caché}$ es el tiempo en que fue guardado en el caché, entonces

$$T_{expiración} = \alpha(T_{guardado\ en\ caché} - T_{última_modificación}) + T_{guardado\ en\ caché}$$

con $\alpha = 0.2$ (este valor se derivó de la experiencia práctica). Hasta $T_{expiración}$, el documento se considera válido y el proxy no se pondrá en contacto con el servidor. Después del tiempo de expi-

ración, el proxy solicita al servidor que envíe una copia reciente, a menos que no haya sido modificado. En otros términos, cuando $\alpha = 0$, la estrategia es igual a la previamente analizada.

Observe que los documentos que no han sido modificados durante mucho tiempo no serán verificados en cuanto a modificaciones tan pronto como aquellos recientemente modificados. La desventaja evidente es que un proxy puede regresar un documento inválido, es decir, un documento más viejo que la versión actual guardada en el servidor. Peor aún, no existe forma de que un cliente detecte que acaba de recibir un documento obsoleto.

Como una alternativa del protocolo basado en extracción es que el servidor notifique a los proxies que un documento ha sido modificado enviando una invalidación. El problema con este método, para los proxies web, es que tal vez el servidor tenga que rastrear un gran número de proxies, lo cual inevitablemente provoca un problema de escalabilidad. Sin embargo, combinando contratos e invalidaciones, Cao y Liu (1998) demostraron que el estado que debe mantenerse en el servidor puede permanecer dentro de límites aceptables. Observemos que este estado es dictado en gran medida por los tiempos de expiración establecidos en los contratos; mientras más bajos son, menos cachés tienen que ser rastreados por un servidor. No obstante, los protocolos de invalidación de cachés de proxies web rara vez se aplican.

Una comparación de políticas de consistencia de almacenamiento en caché en la web se encuentra en Cao y Oszu (2002). Su conclusión es que permitir que el servidor envíe invalidaciones puede abrumar a cualquier otro método en función de ancho de banda y latencia percibida por el cliente, al mismo tiempo que los documentos guardados en caché se mantienen consistentes con aquellos guardados en el servidor de origen. Estos hallazgos se conservan para patrones de acceso como a menudo se observa en el caso de aplicaciones de comercio electrónico.

Otro problema con cachés de proxy web es que sólo pueden ser utilizados para documentos estáticos, es decir, documentos que no son generados al instante por servidores web en respuesta a la solicitud de un cliente. Estos documentos generados dinámicamente a menudo son únicos en el sentido de que la misma respuesta de un cliente, se presume, conducirá a una respuesta diferente la siguiente vez. Por ejemplo, muchos documentos contienen anuncios (llamados **pendones**) que cambian por cada solicitud realizada. Más adelante regresamos a esta situación, cuando abordemos el almacenamiento en caché y la replicación en el caso de aplicaciones web.

Por último, también debemos mencionar que se ha investigado mucho para indagar cuáles son las mejores estrategias de reemplazo de la memoria caché. Existen numerosas propuestas, pero de manera general, las estrategias de reemplazo simples tales como el desalojo del objeto menos recientemente utilizado funcionan lo suficientemente bien. En Podling y Boszormenyi (2003) se encuentra un estudio a fondo acerca de estrategias de reemplazo.

12.6.2 Replicación de sistemas de alojamiento en la web

Como la importancia de la web continúa incrementándose como un vehículo útil para que las organizaciones se presenten a sí mismas e interactúen directamente con los usuarios finales, contemplamos un cambio entre mantener el contenido de un sitio web y asegurarse de que el sitio siga estando fácil y continuamente accesible. Esta distinción ha pavimentado el camino para **redes de entrega de contenido** (**CDN**, por sus siglas en inglés). La idea principal que sirve de fundamento

a estas CDN, es que actúan como servicio de alojamiento en la web, al proporcionar una infraestructura adecuada para distribuir y replicar documentos web de múltiples sitios a través de internet. Su tamaño puede ser impresionante. Por ejemplo, alrededor de 2006, Akamai informó que tenía más de 18 000 servidores esparcidos a través de 70 países.

El enorme tamaño de una CDN requiere que los documentos alojados sean automáticamente distribuidos y replicados, lo que lleva a la arquitectura de un sistema de autogestión tal como vimos en el capítulo 2. En la mayoría de los casos, una CDN a gran escala se organiza a lo largo de las líneas de un lazo de control de retroalimentación, como se muestra en la figura 12-18, y el cual se describe extensamente en Sivasubramanian y colaboradores (2004b).

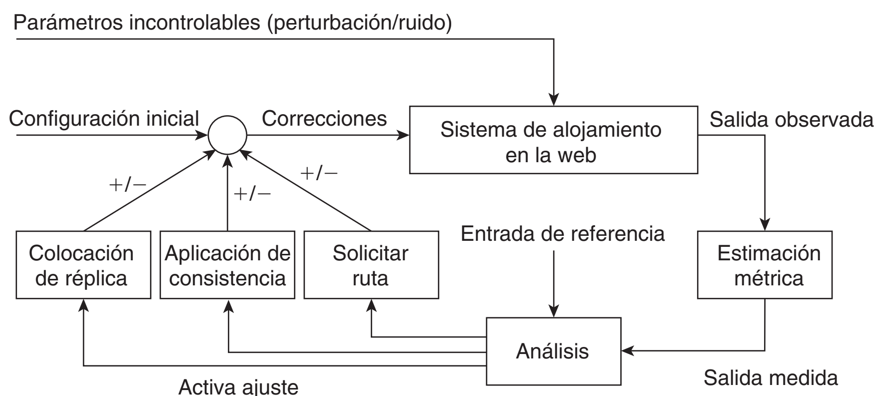


Figura 12-18. Organización general de una CDN como sistema de control de retroalimentación (tomada de Sivasubramanian y cols., 2004b).

En esencia, existen tres clases diferentes de aspectos relacionados con la replicación en sistemas de alojamiento en la web: estimación métrica, activación de una adaptación, y toma de medidas apropiadas. Lo último puede subdividirse en decisiones de colocación de réplicas, hacer que se cumpla la consistencia, y enrutamiento de la solicitud del cliente. A continuación, abordamos brevemente cada uno de estos temas.

Estimación métrica

Un aspecto interesante de las CDN es que necesitan hacer un intercambio entre muchos aspectos cuando se trata de alojar contenido replicado. Por ejemplo, los tiempos de acceso para un documento pueden ser óptimos si está masivamente replicado, pero al mismo tiempo esto implica un costo financiero, así como un costo en función del uso de ancho de banda para diseminar actualizaciones. En general, existen muchas propuestas para estimar qué tan bien está funcionando una CDN. Estas propuestas pueden agruparse en varias clases.

En primer lugar, existe la *métrica de latencia*, mediante la cual se mide el tiempo requerido para realizar una acción; por ejemplo, buscar un documento. Por trivial que parezca, la estimación

de latencias se dificulta cuando, por ejemplo, un proceso que decide sobre la colocación de réplicas tiene que saber la demora que existe entre un cliente y algún servidor remoto. Típicamente, se tendrá que utilizar un algoritmo para determinar el posicionamiento global de nodos como se vio en el capítulo 6.

En lugar de estimar latencia, puede ser más importante medir el ancho de banda disponible entre dos nodos. Esta información es particularmente importante cuando tienen que ser transferidos documentos grandes, ya que en ese caso la capacidad de respuesta del sistema es dictada en gran medida por el tiempo en que un documento puede ser transferido. Existen varias herramientas para medir el ancho de banda disponible, pero en todos los casos resulta que es difícil realizar mediciones precisas. En Strauss y colaboradores (2003) se encuentra más información relacionada con este aspecto.

Otra clase se compone de *métrica espacial*, que consiste principalmente en medir la distancia entre nodos en función del número de conexiones intermedias de enrutamiento a nivel de red, o de conexiones intermedias entre sistemas autónomos. De nueva cuenta, determinar el número de conexiones intermedias entre dos nodos arbitrarios puede ser muy difícil, e incluso puede ser que no esté relacionado con la latencia (Huffaker y cols., 2002). Además, la simple búsqueda en tablas de enrutamiento no va a funcionar cuando se desplieguen técnicas de bajo nivel tales como el **protocolo múltiple de conmutación de etiquetas de** (**MPLS**, por sus siglas en inglés). MPLS evade el enrutamiento a nivel de red mediante el uso de técnicas de circuito virtual para remitir de inmediato y eficientemente paquetes a su destino [vea también Guichard y cols. (2005)]. Los paquetes pueden seguir entonces rutas completamente diferentes de las que aparecen en las tablas de enrutadores a nivel de red.

Una tercera clase está formada por la *métrica de uso de red*, la cual ocasiona con más frecuencia ancho de banda consumido. Calcular el ancho de banda consumido en función del número de bytes que se van a transferir en general es fácil. Sin embargo, para hacerlo correctamente, se debe tomar en cuenta qué tan a menudo se lee el documento, qué tan a menudo se actualiza y con qué frecuencia se replica. Dejamos esto como ejercicio al lector.

La *métrica de consistencia* informa sobre a qué grado se está desviando una réplica de su copia maestra. Ya estudiamos extensamente cómo se puede medir la consistencia en el contexto de consistencia continua en el capítulo 7 (Yu y Vahdat, 2002).

Por último, la *métrica financiera* forma otra clase para medir qué tan bien está funcionando una CDN. Aunque no es una técnica del todo, considerando que la mayor parte de una CDN opera en el ámbito comercial, queda claro que en muchos casos la métrica financiera será decisiva. Además, la métrica financiera está estrechamente relacionada con la infraestructura real de internet. Por ejemplo, la mayoría de las CDN comerciales colocan servidores al margen de internet, es decir, contratan capacidad de ISP para servir directamente a usuarios finales. En este punto, los modelos de negocio se entrelazaron con temas tecnológicos, un área que no está del todo bien entendida. Existe sólo muy poco material disponible sobre la relación entre desempeño financiero y temas tecnológicos (Janiga y cols., 2001).

Estos ejemplos muestran que la medición del desempeño de una CDN, o incluso su estimación, puede ser una tarea extremadamente compleja. En la práctica, para CDN comerciales, el tema que realmente cuenta es si son capaces de satisfacer los acuerdos a nivel de servicio contraídos con el cliente. Estos acuerdos a menudo se formulan simplemente en función de la rapidez con que los

clientes tienen que ser servidos. Asegurarse de que se cumplan estos acuerdos corresponde entonces a la CDN.

Activación de una adaptación

Otra cuestión que debe ser abordada es cuándo y cómo se tienen que activar las adaptaciones. Un modelo simple es estimar periódicamente la métrica y entonces tomar las medidas necesarias. Este método se utiliza a menudo en la práctica. Procesos especiales localizados en los servidores reúnen información y periódicamente buscan cambios.

Una desventaja importante de la evaluación periódica es que los cambios repentinos pueden ser pasados por alto. Un tipo de cambio repentino que está recibiendo considerable atención es el de multitud rápida. Una **multitud rápida** es una ráfaga repentina de solicitudes de un documento web específico. En muchos casos, este tipo de ráfagas puede deprimir por completo un servicio, lo que a su vez puede provocar una cascada de suspensiones temporales del servicio, tal como se ha visto durante varios eventos en la reciente historia de internet.

El manejo de multitud rápida es difícil. Una solución muy cara es replicar masivamente un sitio web en cuanto las tasas de solicitud se incrementen con rapidez, las solicitudes deberán ser redireccionadas hacia las réplicas para reducir la carga de la copia maestra. Este tipo de aprovisionamiento redundante, desde luego, no es la forma de proceder. En cambio, lo que se requiere es un **pronosticador de multitud rápida** que proporcione al servidor el tiempo suficiente como para instalar dinámicamente réplicas de documentos web, después de lo cual puede redireccionar las solicitudes cuando la entrega se complique. Uno de los problemas relacionados con el intento de predecir multitud rápida es que pueden ser muy diferentes. La figura 12-19 muestra algunos trazos de acceso de cuatro sitios web diferentes que experimentaron una multitud rápida. Como punto de referencia, la figura 12-19(a) comprende trazos de acceso que abarcan dos días. También algunas crestas muy acentuadas, aunque por otra parte no está sucediendo nada extraordinario. Por contraste, la figura 12-19(b) muestra un trazo de dos días con cuatro multitudes rápidas. Sigue habiendo algo de regularidad, la cual puede manifestarse después de un rato de modo que puedan tomarse medidas. Sin embargo, el daño pudo haber ocurrido antes de llegar a ese punto.

La figura 12-19(c) muestra un trazo que abarca seis días con, por lo menos, dos multitudes rápidas. En este caso, cualquier pronosticador va a enfrentar un problema serio, ya que ambos incrementos de la tasa de solicitudes ocurrieron casi al mismo tiempo. Finalmente, la figura 12-19(d) muestra una situación en la cual la primera cresta probablemente no provocará adaptaciones, pero la segunda por supuesto que sí. Esta situación resulta ser el tipo de comportamiento que puede ser manejado bastante bien mediante un análisis en tiempo de ejecución.

Un método promisorio para predecir multitud rápida es utilizar una técnica de extrapolación simple. Baryshikov y colaboradores (2005) proponen medir continuamente el número de solicitudes de un documento durante un intervalo de tiempo específico $[t - W, t)$, donde W es el **tamaño de la ventana**. El propio intervalo se divide en lapsos pequeños, donde por cada lapso se cuenta el número de solicitudes. Entonces, aplicando una regresión lineal simple, podemos ajustar una curva f_t que exprese el número de accesos en función del tiempo. Extrapolando la curva a instancias de

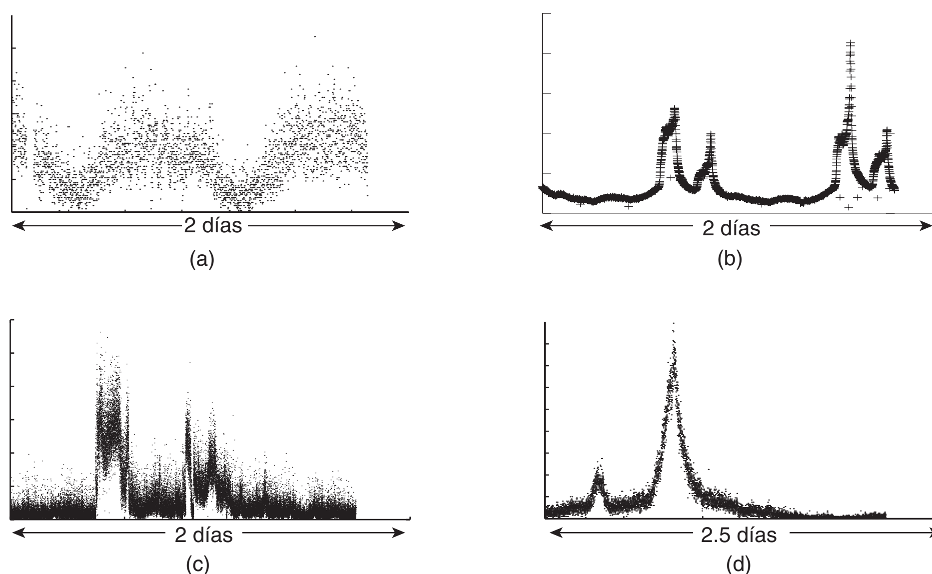


Figura 12-19. Un patrón de acceso normal y tres patrones de acceso diferentes que reflejan el comportamiento de una multitud rápida (tomado de Baryshnikov y cols., 2005).

tiempo más allá de *t*, obtenemos un pronóstico del número de solicitudes. Cuando se pronostica que el número de solicitudes excederá un umbral dado, se activa una alarma.

Este método funciona notablemente bien con patrones de acceso múltiple. Por desgracia, la determinación del tamaño de la ventana y del umbral de la alarma depende mucho del tráfico existente en el servidor web. En la práctica, esto significa que se requiere mucha afinación final manual para configurar un pronosticador ideal para un sitio específico. Aún no se sabe cómo pueden ser configurados automáticamente los pronosticadores de multitud rápida.

Medidas de ajuste

Como ya mencionamos, esencialmente existen sólo tres medidas (relacionadas) que se pueden tomar para cambiar el comportamiento de un servicio de alojamiento en la web: cambiar la colocación de réplicas, cambiar la observancia forzosa de la consistencia, y decidir sobre cómo y cuándo redireccionar solicitudes de los clientes. Ya estudiamos extensamente las primeras dos medidas en el capítulo 7. El redireccionamiento de solicitudes de los clientes merece más atención. Antes de analizar algunos de los intercambios, veamos primero cómo se manejan la consistencia y la replicación en la práctica considerando la situación Akamai (Leighton y Lewin, 2000; y Dilley y cols., 2002).

La idea básica es que cada documento web se componga de una página HTML (o XML) principal donde algunos otros documentos tales como imágenes, video y audio han sido incrustados.

Para desplegar todo el documento, es necesario que el navegador del usuario también busque los documentos embebidos. La suposición es que los documentos embebidos rara vez cambian, por lo cual tiene sentido guardarlos en caché o replicarlos.

La referencia a un documento embebido normalmente se hace por medio de un URL. Sin embargo, en la CDN de Akamai, el URL se modifica de modo que se refiera a un **fantasma virtual**, el cual es una referencia a un servidor en la CDN. El URL también contiene el nombre del servidor del servidor de origen por razones que se explican a continuación. El URL modificado se resuelve como sigue, y también se muestra en la figura 12-20.

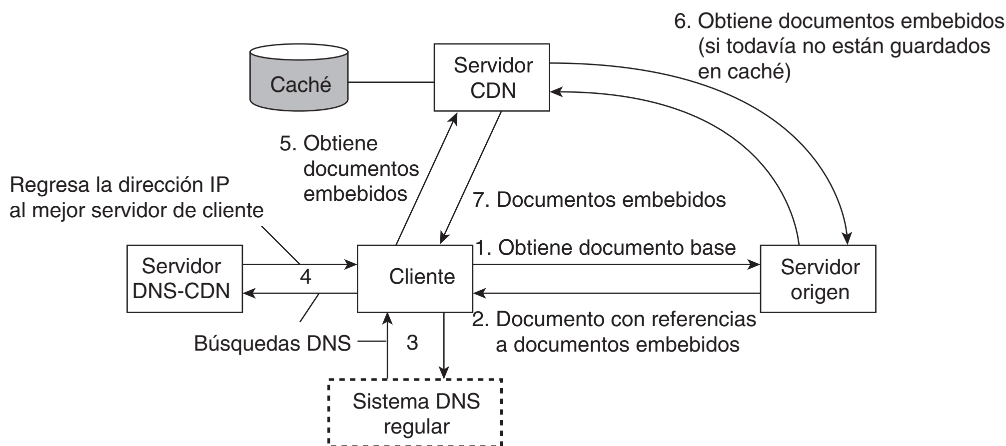


Figura 12-20. Funcionamiento principal de la CDN Akamai.

El nombre de fantasma virtual incluye un nombre DNS tal como *ghosting.com*, el cual es transformado por el sistema de asignación de nombres DNS en un servidor DNS CDN (el resultado del paso 3). Cada servidor DNS rastrea los servidores cercanos al cliente. Con esta finalidad, cualquiera de las métricas de proximidad previamente analizadas podría ser utilizada. En efecto, los servidores DNS CDN redireccionan al cliente hacia un servidor replicado que sea mejor para dicho cliente (paso 4), que podría ser el más cercano, el menos cargado, o una combinación de varias métricas (la política de redireccionamiento existente está patentada).

Por último, el cliente remite la solicitud del documento embebido al servidor CDN seleccionado. Si este servidor aún no tiene el documento, lo busca en el servidor original (mostrado como paso 6), lo guarda localmente en caché y posteriormente lo transfiere al cliente. Si el documento ya estaba en el caché del servidor CDN, puede ser regresado de inmediato. Observemos que para buscar el documento embebido, el servidor réplica debe ser capaz de enviar una solicitud al servidor origen, razón por la cual su nombre de servidor también está contenido en el URL del documento embebido.

Un aspecto interesante de este esquema es la simplicidad mediante la cual se puede hacer que se cumpla la consistencia de los documentos. Claramente, siempre que un documento principal

cambie, un cliente será capaz de obtenerlo del servidor origen. En el caso de documentos embebidos, se debe seguir un método diferente ya que, en principio, estos documentos se obtienen de un servidor réplica cercano. Con esta finalidad, un URL de un documento embebido no sólo se refiere a un nombre de servidor especial que eventualmente conduce a un servidor DNS CDN, sino que también contiene un identificador único que cambia cada vez que cambia el documento embebido. En efecto, este identificador cambia el nombre del documento embebido. Por consiguiente, cuando el cliente es redireccionado hacia un servidor CDN específico, éste no encontrará el documento nombrado en su caché y, por tanto, lo obtendrá del servidor origen. El documento viejo finalmente será eliminado del caché del servidor ya que no se hará referencia a él.

Este ejemplo muestra la importancia de redireccionar las solicitudes de los clientes. En principio, redireccionando apropiadamente a los clientes, una CDN puede permanecer en control en cuanto al desempeño percibido por el cliente, pero también al tomar en cuenta el desempeño total del sistema evitando, por ejemplo, que las solicitudes sean enviadas hacia servidores excesivamente cargados. Estas llamadas **políticas de redireccionamiento adaptables** se aplican cuando se proporciona información sobre el comportamiento actual del sistema a procesos que toman decisiones de redireccionamiento. Esto hace que se regrese en parte a las técnicas de estimación métrica previamente analizadas.

Además de las diferentes políticas, un tema importante es si el redireccionamiento de solicitudes es transparente o no para el cliente. En esencia, existen sólo tres técnicas de redireccionamiento: transferencia de TCP, redireccionamiento de DNS y redireccionamiento de HTTP. Ya estudiamos la transferencia de TCP. Esta técnica se aplica sólo a grupos de servidores y no crece a redes de área ancha.

El redireccionamiento de DNS es un mecanismo transparente mediante el cual un cliente puede permanecer completamente ajeno a la localización de los documentos. El redireccionamiento de dos niveles de Akamai es un ejemplo de esta técnica. También se puede aplicar DNS directamente para regresar una de varias direcciones, tal como ya vimos. Observe, sin embargo, que el redireccionamiento de DNS se aplica sólo a un sitio completo: el nombre de documentos individuales no cabe en el espacio de nombre DNS.

El redireccionamiento de HTTP, finalmente, es un mecanismo no transparente. Cuando un cliente solicita un documento específico, se le puede dar un URL alternativo como parte del mensaje de respuesta HTTP al cual luego es redireccionado. Una observación importante es que este URL es visible para el navegador del cliente. De hecho, el usuario puede decidir marcar el URL referido, lo que potencialmente inutiliza la política de redireccionamiento.

12.6.3 Replicación de aplicaciones web

Hasta este punto hemos abordado, principalmente, el almacenamiento en caché y la replicación estática del contenido web. En la práctica, vemos que la web ofrece cada vez más contenido dinámicamente generado, pero que también se está expandiendo hacia la oferta de servicios que pueden ser invocados por aplicaciones remotas. En estas situaciones, también advertimos que el almacenamiento en caché y la replicación pueden ayudar considerablemente a mejorar el desempeño total, aunque los métodos para lograrlo son más sutiles que los estudiados hasta ahora [vea también Conti y cols. (2005)].

Cuando se considera mejorar el desempeño de aplicaciones web mediante almacenamiento en caché y replicación, las cosas se complican por el hecho de que varias soluciones pueden ser desplegadas, sin que ninguna sobresalga como la mejor. Analicemos la situación de servidor periférico ilustrada en la figura 12-21. En este caso, suponemos que en cada sitio alojado una CDN tiene un servidor origen que actúa como sitio autoritario para todas las operaciones de lectura y actualización. Un servidor periférico se utiliza para manejar solicitudes de clientes y tiene la habilidad de guardar información (parcial) como también está guardada en el servidor origen.

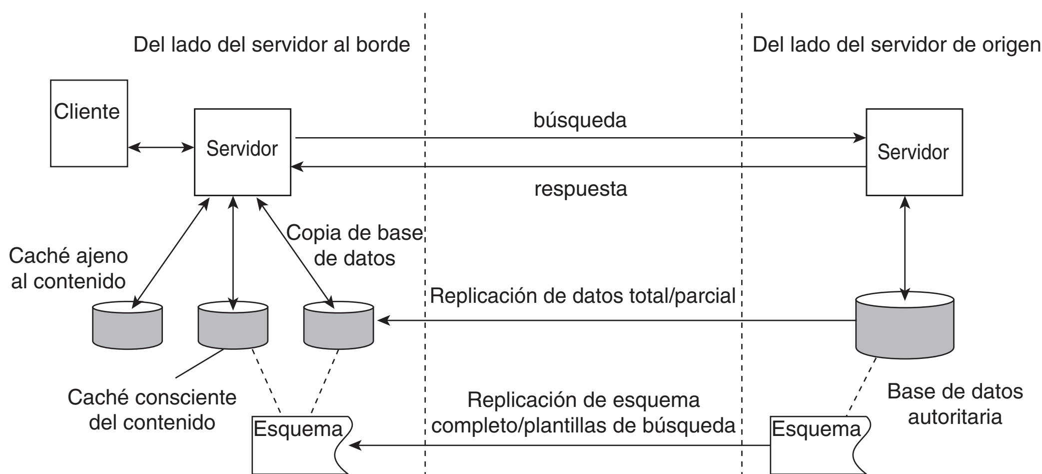


Figura 12-21. Alternativas de almacenamiento en caché y replicación con aplicaciones web.

Recordemos que en una arquitectura de servidor periférico, los clientes web solicitan datos mediante un servidor periférico, el que a su vez obtiene su información del servidor origen asociado con el sitio web al que se refiere el cliente. Como también se muestra en la figura 12-21, se supone que el servidor origen está compuesto por una base de datos con la cual se crean respuestas dinámicamente. Aunque se ha mostrado sólo un servidor web, es común organizar cada servidor de acuerdo con una arquitectura de varios niveles como ya vimos. Un servidor periférico puede entonces ser aproximadamente organizado como sigue.

En primer lugar, para mejorar el desempeño, podemos decidir aplicar una replicación completa de los datos guardados en el servidor origen. Este esquema funciona bien siempre que la proporción de actualización sea baja y cuando las búsquedas requieren una extensa investigación en la base de datos. Como ya mencionamos, se supone que todas las actualizaciones se realizan en el servidor origen, el cual asume la responsabilidad de mantener las réplicas y los servidores al borde en un estado consistente. Las operaciones de lectura pueden, por tanto, realizarse en los servidores al borde. Aquí observamos que la replicación en busca de desempeño fallará cuando la proporción de actualización sea muy alta, ya que cada actualización incurrirá en comunicación a través de una red

de área amplia para llevar las réplicas a un estado consistente. Como se muestra en Sivasubramanian y colaboradores (2004a), la relación lectura-actualización es el factor que determina a qué grado debe ser replicada la base de datos origen en un entorno de área amplia.

Otro caso de replicación completa es cuando las búsquedas en general son complejas. En el caso de una base de datos relacional, esto significa que una búsqueda requiere que se revisen y procesen múltiples tablas, como generalmente sucede con una operación "join". Opuestas a las búsquedas complejas están las búsquedas simples que, por lo común, requieren acceso a una sola tabla para producir una respuesta. En el último caso, puede ser suficiente una **replicación parcial**, mediante la cual un subconjunto de datos se guarda en el servidor periférico.

El problema con la replicación parcial es que puede resultar muy difícil decidir manualmente qué datos se requieren en el servidor periférico. Sivasubramanian y colaboradores (2005) proponen manejar esto automáticamente al replicar registros de acuerdo con el mismo principio que Globule replica sus páginas web. Como ya estudiamos en el capítulo 2, esto significa que un servidor origen analiza trazos de acceso a registros de datos en los que posteriormente basa su decisión sobre dónde colocar los registros. Recordemos que en Globule, la toma de decisiones se basaba en el costo de ejecutar operaciones de lectura y escritura una vez que los datos estaban en su lugar (y posiblemente replicados). Los costos se expresan en una función lineal simple:

$$\text{costo} = (w_1 \times m_1) + (w_2 \times m_2) + \cdots + (w_n \times m_n)$$

donde m_k es la métrica de desempeño (tal como ancho de banda consumido), y $w_k > 0$ es el peso relativo que indica qué tan importante es la métrica.

Una alternativa de la replicación parcial es utilizar **cachés conscientes del contenido**. La idea básica en este caso es que un servidor periférico mantenga una base de datos local adaptada ahora al tipo de búsquedas que pueden ser manejadas por el servidor origen. Para explicarlo, en un sistema de base de datos completa, una búsqueda operará en una base de datos donde los datos están organizados en tablas de tal suerte que, por ejemplo, la redundancia se reduzca al mínimo. Las bases de datos de este tipo se conocen también como **normalizadas.**

En esas bases de datos, cualquier búsqueda que se apegue al esquema de datos, en principio, puede ser procesada, aunque probablemente a costos considerables. Con cachés conscientes del contenido, un servidor periférico mantiene una base de datos que está organizada de acuerdo con la estructura de búsquedas. Lo que esto significa es que suponemos que las búsquedas se adhieren a un limitado número de plantillas, así que las diferentes clases de búsqueda que pueden ser procesadas se restringen. En estos casos, sin embargo, siempre que se recibe una búsqueda, el servidor periférico la compara con las plantillas disponibles y posteriormente busca en su base de datos local para componer una respuesta, si es posible. Si los datos solicitados no están disponibles, la búsqueda se remite al servidor origen, después de lo cual la respuesta se guarda en caché antes de regresarla al cliente.

En realidad, lo que hace el servidor al borde es verificar si una búsqueda puede ser respondida con los datos guardados localmente. Esto también se conoce como **verificación de contención de búsqueda**. Este tipo de datos se almacena localmente como respuesta a una petición realizada anteriormente. Este método funciona mejor cuando las búsquedas tienden a repetirse.

Parte de la complejidad del almacenamiento en caché consciente del contenido se deriva de que en el servidor al borde los datos tienden a mantenerse consistentes. Con esta finalidad, el servidor origen debe saber qué registros están asociados con cuáles plantillas, de modo que cualquier actualización de un registro o de una tabla pueda ser abordada apropiadamente, por ejemplo, enviando un mensaje de invalidación a los servidores al borde pertinentes. Otra causa de complejidad proviene de que las búsquedas aún tienen que ser procesadas en servidores al borde. En otros términos, no es insignificante el poder de cómputo necesario para manejar búsquedas. Considerando que las bases de datos a menudo constituyen un cuello de botella para el desempeño óptimo en servidores web, puede ser que se requieran soluciones alternativas. Por último, el almacenamiento en caché resultante de búsquedas que implican varias tablas (es decir, cuando las búsquedas son complejas), de modo que la verificación de una contención de búsqueda pueda ser realizada efectivamente, no es trivial. La razón es que la organización de los resultados puede ser muy diferente de la organización de las tablas en las que operó la búsqueda.

Estas observaciones conducen a una tercera solución, denominada **almacenamiento en caché ajeno al contenido**, que se describe con todo detalle en Sivasubramanian y colaboradores (2006). La idea de almacenamiento en caché ajeno al contenido es extremadamente simple: cuando un cliente entrega una búsqueda a un servidor al borde, el servidor primero calcula un valor disperso único para dicha búsqueda. Con este valor disperso, posteriormente busca en su caché para ver si ya antes ha procesado tal búsqueda. Si no, la búsqueda se remite al origen y el resultado se guarda en caché antes de regresarlo al cliente. Si la búsqueda ya fue procesada, el resultado previamente guardado en caché se regresa al cliente.

La ventaja principal de este esquema es el reducido esfuerzo computacional que se requiere de un servidor al borde en comparación con los métodos de base de datos descritos anteriormente. Sin embargo, el almacenamiento en caché ajeno al contenido puede ser inútil en función de almacenamiento, ya que los cachés pueden contener muchos más datos redundantes en comparación con el almacenamiento en caché consciente del contenido o la replicación de base de datos. Observe que la redundancia también complica el proceso de conservar el caché actualizado puesto que el servidor origen puede necesitar llevar una cuenta precisa de qué actualizaciones pueden afectar potencialmente los resultados de búsqueda guardados en caché. Estos problemas se aligeran cuando se asume que las búsquedas pueden coincidir con sólo un conjunto limitado de plantillas predefinidas, tal como lo analizamos previamente.

Desde luego, estas técnicas pueden ser igualmente desplegadas para la generación venidera de servicios web, pero aún hay mucho que investigar antes de poder identificar soluciones estables.

12.7 TOLERANCIA A FALLAS

En los sistemas distribuidos basados en la web, la tolerancia a fallas se logra principalmente por medio de almacenamiento en caché del lado del cliente y replicación de servidores. Ningún método especial se incorpora a, por ejemplo, HTTP para promover la tolerancia a fallas o la recuperación después de ocurrir éstas. Observemos, sin embargo, que la alta disponibilidad en la web se logra mediante redundancia que utiliza técnicas generalmente disponibles en servicios cruciales como

DNS. Como un ejemplo que ya se mencionó, DNS permite que varias direcciones sean regresadas como resultado de la búsqueda de un nombre. En sistemas basados en la web tradicionales, la tolerancia a fallas puede ser relativamente fácil de implementar considerando el diseño de servidores sin estado, junto con la frecuente naturaleza estática del contenido provisto.

Cuando se trata de servicios web, observaciones similares son válidas: difícilmente se introducen técnicas nuevas o especiales que tengan que ver con fallas (Birman, 2005). Sin embargo, debe quedar claro que los problemas de ocultar fallas y recuperaciones pueden ser más severos. Por ejemplo, los servicios web que soportan transacciones y soluciones distribuidas en un área amplia definitivamente tendrán que enfrentar servicios participantes que fallan o comunicación no confiable.

Aún más importante es que, en el caso de servicios web, podemos fácilmente encontrarnos con grafos de invocación complejos. Observemos que en muchos sistemas basados en la web, el cálculo sigue una simple convención de invocación cliente-servidor de dos niveles. Esto significa que un cliente llama a un servidor, el cual enseguida calcula una respuesta sin la necesidad de utilizar servicios adicionales externos. Como ya mencionamos, la tolerancia a fallas a menudo se puede lograr replicando simplemente el servidor o confiando en parte en el resultado guardado en caché.

Esta situación ya no es válida para servicios web. En muchos casos, ahora tratamos con soluciones de varios niveles donde los servidores también actúan como clientes. Aplicar replicación a servidores significa que los clientes e invocadores tienen que manejar invocaciones replicadas, justo como en el caso de objetos replicados que presentamos en el capítulo 10.

Los problemas se agravan para servicios diseñados para manejar fallas bizantinas. La replicación de componentes desempeña un rol crucial en este caso, pero también lo hace el protocolo que ejecutan los clientes. Además, ahora se tiene que enfrentar la situación de que un servicio tolerante a fallas bizantinas (**BFT**, por sus siglas en inglés) puede necesitar actuar como cliente de otro servicio no replicado. Una solución a este problema es propuesta por Merideth y cols. (2005) que se basa en el sistema BFT, propuesto por Castro y Liskov (2002), la cual estudiamos en el capítulo 11.

Existen tres temas que deben ser manejados. En primer lugar, los clientes de un servicio BFT deberán ver al servicio como a cualquier otro servicio web. En particular, esto significa que la replicación interna de dicho servicio deberá quedar oculta para el cliente, junto con un procesamiento de respuestas apropiado. Por ejemplo, un cliente necesita reunir $k + 1$ respuestas idénticas de hasta $2k + 1$ respuestas, suponiendo que el servicio BFT está diseñado para manejar cuando mucho k procesos con fallas. Típicamente, este tipo de procesamiento de respuesta puede ocultarse en resguardos del lado del cliente, las cuales pueden ser automáticamente generadas a partir de aplicaciones WSDL.

En segundo lugar, un servicio BFT deberá garantizar la consistencia interna cuando actúa como cliente. En particular, tiene que manejar el caso de que el servicio externo al que está llamando regrese diferentes respuestas a diferentes réplicas. Esto podría suceder, por ejemplo, cuando el propio servicio externo falla por cualquier razón. Por consiguiente, es probable que las réplicas tengan que ejecutar un protocolo de acuerdo adicional como extensión de los protocolos que ya están ejecutando para proporcionar tolerancia a fallas bizantinas. Después de ejecutar este protocolo, pueden regresar sus respuestas al cliente.

Por último, los servicios externos también deberán tratarse como un servicio BFT que actúa como cliente y como una sola entidad. En particular, un servicio simplemente no puede aceptar una solicitud que viene de una sola réplica, pero puede proceder sólo cuando ha recibido por lo menos $k + 1$ solicitudes idénticas de réplicas diferentes.

Existen tres situaciones que conducen a tres piezas diferentes de software que deben ser integradas en juegos de herramientas para desarrollar servicios web. Detalles y evaluaciones de su desempeño se encuentran en Merideth y colaboradores (2005).

12.8 SEGURIDAD

Si consideramos la naturaleza abierta de internet, idear una arquitectura de seguridad que proteja a clientes y servidores contra diversos ataques es crucialmente importante. La mayoría de los temas sobre seguridad en la web tienen que ver con el establecimiento de un canal seguro entre un cliente y un servidor. El método predominante para establecer un canal seguro en la web es utilizar una **capa de socket seguro** (**SSL**, por sus siglas en inglés), la cual originalmente fue implementada por Netscape. Aunque la SSL nunca ha sido formalmente estandarizada, la mayoría de los clientes y servidores web le dan soporte. Una actualización de SSL ha sido formalmente presentada en RFC 2246 y RFC 3546, y ahora es conocida como protocolo de **seguridad en la capa de transporte** (**TLS**, por sus siglas en inglés) (Dierks y Allen, 1996; y Blake-Wilson y cols., 2003).

Como se muestra en la figura 12-22, TLS es un protocolo de seguridad independiente de la aplicación y lógicamente colocado sobre el protocolo de transporte. Por razones de simplicidad, las implementaciones de TLS (y SSL) generalmente se basan en TCP. TLS puede soportar gran diversidad de protocolos de alto nivel, incluido el HTTP, como analizamos a continuación. Por ejemplo, es posible implementar versiones seguras de FTP o Telnet por medio de TLS.

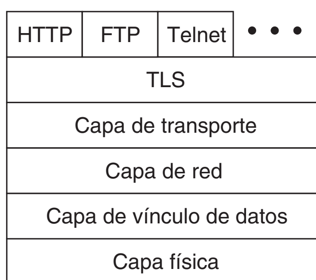


Figura 12-22. Posición del protocolo TLS en la pila de protocolos de internet.

El propio TLS está organizado en dos capas. El núcleo del protocolo está formado por la **capa de protocolo de registro TLS**, la cual implementa un canal seguro entre un cliente y un servidor. Las características exactas del canal se determinan durante su establecimiento, aunque pueden incluir fragmentación y compresión de mensajes, las cuales se aplican junto con la autentificación, la integridad, y la confidencialidad de los mensajes.

El establecimiento de un canal seguro se hace en dos fases, como se muestra en la figura 12-23. En primer lugar, el cliente informa al servidor acerca de los algoritmos criptográficos que puede manejar, así como también de cualquier método de compresión que soporte. La selección en sí es hecha siempre por el servidor, el cual informa sobre su selección al cliente. Estos dos primeros mensajes se muestran en la figura 12-23.

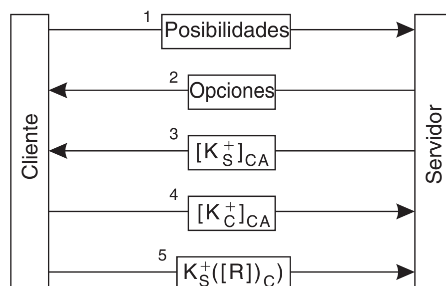


Figura 12-23. Protocolo TLS con autentificación mutua.

En la segunda fase, ocurre la autentificación. Siempre se requiere que el servidor se autentifique a sí mismo, razón por la cual pasa al cliente un certificado que contiene su clave pública firmada por una autoridad de certificación *AC*. Si el servidor requiere que el cliente se autentifique, también el cliente deberá enviar un certificado al servidor, mostrado como el mensaje 4 en la figura 12-23.

El cliente genera un número aleatorio que será utilizado por ambos lados para construir una clave de sesión, y envía este número al servidor, cifrada con la clave pública del servidor. Además, si se requiere autentificación del cliente, éste firma el número con su clave privada, lo cual conduce al mensaje 5 en la figura 12-23. (En realidad, aparte es enviado un mensaje con una versión cifrada y firmada del número aleatorio, produciendo el mismo efecto.) En ese punto, el servidor puede verificar la identidad del cliente, después de lo cual se establece el canal seguro.

12.9 RESUMEN

Se puede argumentar que los sistemas distribuidos basados en la web han hecho que las aplicaciones en red lleguen a ser populares con los usuarios finales. El uso de la noción de un documento web como forma de intercambiar información se acerca a la forma en que las personas con frecuencia se comunican en ambientes de oficina y en otros entornos. Todo mundo entiende lo que es un documento escrito en papel, así que la extensión de este concepto a documentos electrónicos es bastante lógica para la mayoría de las personas.

El soporte de hipertexto provisto a los usuarios finales de la web ha sido de primordial importancia para alcanzar su popularidad. Además, los usuarios finales por lo común ven una arquitectura

cliente-servidor en donde los documentos simplemente son obtenidos de un sitio específico. Sin embargo, los sitios web modernos están organizados a lo largo de arquitecturas de varios niveles donde un componente final sólo es responsable de generar páginas HTML o XML como respuestas que pueden ser presentadas al cliente.

El reemplazo de un usuario final con una aplicación ha traído más servicios web. Desde un punto de vista tecnológico, los servicios web por sí mismos en general no son espectaculares, pero aún están en desarrollo. Lo importante, sin embargo, es que servicios muy diferentes tienen que ser descubiertos y puestos a la disposición de clientes autorizados. Por consiguiente, se han hecho enormes esfuerzos para estandarizar las descripciones de los servicios, las comunicaciones, los directorios y varias interacciones. De nueva cuenta, cada estándar por sí mismo no representa particularmente nuevas ideas, pero por ser un estándar contribuye a la expansión de servicios web.

En la web, los procesos están diseñados para manejar solicitudes HTTP, de los cuales el servidor web Apache es un ejemplo canónico. Apache ha demostrados ser un vehículo versátil para manejar servicios basados en HTTP, pero también puede ser fácilmente ampliado para que satisfaga ciertas necesidades específicas tales como replicación.

Como la web opera a través de internet, se ha prestado mucha atención a mejorar el desempeño mediante almacenamiento en memoria caché y la replicación. Se han desarrollado técnicas más o menos estándar para almacenamiento en caché del lado del cliente, pero cuando se trata de replicación se han logrado avances considerables. De manera notable, cuando la replicación de aplicaciones web está en riesgo, resulta que diferentes soluciones tendrán que coexistir para lograr el desempeño óptimo.

Tanto la tolerancia a fallas como la seguridad en general se manejan por medio de técnicas estándar que se han aplicado desde hace mucho tiempo en muchos otros sistemas distribuidos.

PROBLEMAS

1. ¿A qué grado el correo electrónico forma parte del modelo de documento web?

2. En muchos casos, los sitios web están diseñados para ser accesados por usuarios. Sin embargo, cuando se trata de servicios web, observamos que los sitios web se vuelven dependientes unos de otros. Considerando la arquitectura de tres niveles de la figura 12-3, ¿dónde esperaría usted ver que ocurra la dependencia?

3. La web utiliza un método basado en archivos a documentos mediante el cual un cliente obtiene primero un archivo antes de abrirlo y desplegarlo. ¿Cuál es la consecuencia de este enfoque en el caso de archivos multimedia?

4. Se podría argumentar que desde un punto de vista tecnológico, los servicios web no abordan nuevos temas. ¿Cuál es el argumento apremiante para considerar importantes los servicios web?

5. ¿Cuál sería la ventaja principal de utilizar el servidor distribuido que analizamos en el capítulo 3 para implementar un cluster de servidores web, en comparación con la forma en que dichos clústeres están organizados como se muestra en la figura 12-9. ¿Cuál es una desventaja evidente?

6. ¿Por qué las conexiones persistentes mejoran el desempeño en general comparadas con las no persistentes?

7. A menudo se dice que SOAP se apega a la semántica RPC. ¿Es realmente cierto eso?

8. Explique la diferencia que hay entre un plug-in, un applet, un servlet y un programa CGI.

9. En WebDAV, ¿es suficiente con que un cliente muestre sólo la señal de bloqueo al servidor para obtener permisos de escritura?

10. En lugar de permitir que un proxy web calcule un tiempo de expiración para un documento, un servidor podría hacerlo. ¿Cuál sería el beneficio de semejante método?

11. Con páginas web que se vuelven altamente personalizadas (porque pueden ser generadas en forma dinámica según la demanda de cada cliente) se podría argumentar que los cachés web pronto serán obsoletos. Aunque es muy probable que esto no suceda en un futuro inmediato. Explique por qué.

12. ¿Sigue la CDN Akamai un protocolo de distribución basado en extracción o en adición?

13. Describa un esquema simple mediante el cual un servidor CDN Akamai pueda indagar que un documento embebido guardado en caché es obsoleto sin verificar la validez del documento en el servidor original.

14. ¿Tendría sentido asociar una estrategia de replicación con cada documento web por separado, en oposición a utilizar sólo unas cuantas estrategias globales?

15. Suponga que un documento no replicado de s bytes es solicitado r veces por segundo. Si el documento se replica a k servidores diferentes y, asumiendo que las actualizaciones se propagan por separado hacia cada réplica, ¿en qué momento será más barata la replicación que cuando el documento no está replicado?

16. Considere que un sitio web está experimentando una multitud rápida. ¿Cuál podría ser una medida apropiada para garantizar que los clientes sigan estando bien atendidos?

17. Existen, en principio, tres técnicas diferentes de redireccionar clientes a servidores: transferencia de TCP, redirección basada en DNS y redirección basada en HTTP. ¿Cuáles son las ventajas y desventajas principales de cada técnica?

18. Proporcione un ejemplo donde una verificación de contención de búsqueda tal como la realiza un servidor al borde que soporta almacenamiento en caché consciente del contenido regresará con éxito.

19. (**Tarea para el laboratorio.**) Establezca un sistema simple basado en la web instalando y configurando el servidor web Apache para su máquina local de modo que pueda ser accesado con un navegador local. Si tiene múltiples computadoras en una red de área local, asegúrese de que el servidor pueda ser accesado con cualquier navegador instalado en dicha red.

20. (**Tarea para el laboratorio.**) WebDAV es soportado por el servidor web Apache y permite que múltiples usuarios compartan archivos para lectura y escritura a través de internet. Instale y configure el servidor Apache para un directorio habilitado con WebDAV en una red de área local. Pruebe la configuración utilizando un cliente WebDAV.

13

SISTEMAS DISTRIBUIDOS BASADOS EN COORDINACIÓN

En los capítulos previos abordamos diferentes formas de ver los sistemas distribuidos, y cada capítulo se enfocó en un solo tipo de datos como base para la distribución. El tipo de datos, ya sea un objeto, un archivo, o un documento (web), tiene sus orígenes en sistemas no distribuidos. Se adaptó para sistemas distribuidos de tal forma que muchos temas relacionados con la distribución pudieran hacerse transparentes para los usuarios y los desarrolladores.

En este capítulo consideramos una generación de sistemas distribuidos que asumen que los diversos componentes de un sistema están inherentemente distribuidos y que el problema real en el desarrollo de tales sistemas radica en coordinar las actividades de distintos componentes. En otros términos, en lugar de concentrarse en la distribución transparente de los componentes, el énfasis radica en la coordinación de actividades entre dichos componentes.

Veremos que algunos aspectos de coordinación ya se han abordado en los capítulos previos, en especial cuando se consideraron los sistemas basados en eventos. Como ha sucedido, muchos sistemas distribuidos convencionales gradualmente han estado incorporando mecanismos que desempeñan un rol fundamental en los sistemas basados en coordinación.

Antes de dar un vistazo a ejemplos prácticos de sistemas, presentamos brevemente la noción de coordinación en sistemas distribuidos.

13.1 INTRODUCCIÓN A LOS MODELOS DE COORDINACIÓN

Para el método que se sigue en sistemas basados en coordinación, es fundamental establecer una clara separación entre cómputo y coordinación. Si vemos a un sistema distribuido como un conjunto

de procesos (posiblemente de multihilos), entonces la parte de cómputo de un sistema distribuido está formada por los procesos, y cada proceso se ocupa en efectuar una actividad computacional específica la cual, en principio, es realizada independientemente de las actividades de otros procesos.

En este modelo, la parte de coordinación de un sistema distribuido maneja la comunicación y la cooperación entre procesos. Forma el pegamento que aglutina las actividades realizadas por los procesos en un todo (Gelernter y Carriero, 1992). En sistemas distribuidos basados en coordinación, el enfoque recae en cómo ocurre la coordinación entre los procesos.

Cabri y colaboradores (2000) proporcionan una taxonomía de modelos de coordinación de agentes móviles que pueden ser aplicados por igual a muchos otros tipos de sistemas distribuidos. Adaptando su terminología a los sistemas distribuidos en general, realizamos una distinción entre modelos a lo largo de dos dimensiones diferentes, temporales y referenciales, como se muestra en la figura 13-1.

		Temporal	
		Acoplado	**Desacoplado**
Referencial	**Acoplado**	Directa	Buzón de correo
	Desacoplado	Orientada a la reunión	Comunicación generativa

Figura 13-1. Taxonomía de modelos de coordinación (adaptada de Cabri y cols., 2000).

Cuando los procesos se acoplan temporal y referencialmente, la coordinación ocurre de una forma directa, conocida como **coordinación directa**. El acoplamiento referencial aparece por lo común en la forma de hacer referencia explícita en comunicación. Por ejemplo, un proceso puede comunicarse sólo si conoce el nombre o identificador del otro proceso con el que desea intercambiar información. Acoplamiento temporal significa que los dos procesos que se están comunicando están funcionando. Este acoplamiento es análogo a la comunicación orientada a mensajes transitorios estudiada en el capítulo 4.

Un tipo diferente de comunicación ocurre cuando los procesos se desacoplan temporalmente, pero se acoplan referencialmente, lo cual se conoce como **coordinación de buzón de correo**. En este caso, no se requiere que los dos procesos que se están comunicando funcionen al mismo tiempo para que ocurra la comunicación. En cambio, la comunicación ocurre colocando mensajes en un buzón de correo (posiblemente compartido). Esta situación es análoga a la comunicación persistente orientada a mensajes descrita en el capítulo 4. Es necesario referirse explícitamente al buzón de correo que mantendrá los mensajes a ser intercambiados. Por consiguiente, el acoplamiento es referencial.

La combinación de sistemas referencialmente desacoplados y temporalmente acoplados forman el grupo de modelos de **coordinación orientada a la reunión**. En sistemas referencialmente desacoplados, los procesos no se conocen explícitamente entre sí. En otros términos, cuando un proceso desea coordinar sus actividades con otros procesos, no puede referirse directamente a otro

proceso. En cambio, existe un concepto de reunión en el cual los procesos se agrupan temporalmente para coordinar sus actividades. El modelo prescribe que los procesos de reunión se ejecuten al mismo tiempo.

Los sistemas basados en reunión a menudo se implementan por medio de eventos como los soportados por sistemas distribuidos basados en objetos. En este capítulo, presentamos otro mecanismo útil para implementar reuniones denominado **sistemas de publicación y suscripción**. En estos sistemas, los procesos pueden suscribirse a mensajes que contienen información sobre temas específicos, en tanto que otros procesos producen (es decir, publican) tales mensajes. La mayoría de los sistemas de publicación y suscripción requiere que los procesos que se están comunicando estén activos al mismo tiempo; por consiguiente, existe un acoplamiento temporal. Sin embargo, en caso contrario, los procesos que se están comunicando pueden permanecer anónimos.

El modelo de coordinación más ampliamente conocido es la combinación de procesos referencial y temporalmente desacoplados, el cual se ejemplifica mediante **comunicación generativa** tal como se presenta en el sistema de comunicación Linda de Gelernter (1985). La idea fundamental en comunicación generativa es que un conjunto de procesos independientes utilicen un espacio de datos persistentes compartidos de tuplas. Las **tuplas** son registros de datos etiquetados compuestos a partir de varios (pero posiblemente cero) campos de entrada desde el teclado. Los procesos pueden poner cualquier tipo de registro en el espacio de datos compartidos (es decir, generan registros de comunicación). A diferencia del caso con pizarrones, no existe la necesidad de ponerse de acuerdo de antemano sobre la estructura de las tuplas. La etiqueta sólo se utiliza para distinguir entre las tuplas que representan diferentes clases de información.

Una característica interesante de estos espacios de datos compartidos es que implementan un mecanismo de búsqueda asociativa de tuplas. En otros términos, cuando un proceso desea extraer una tupla del espacio de datos, esencialmente especifica (algunos de) los valores de los campos en que está interesado. Cualquier tupla que coincida con esa especificación es retirado entonces del espacio de datos y transferido al proceso. Si no se encuentra ninguna coincidencia, el proceso puede elegir bloquearse hasta que haya una tupla coincidente. Los detalles de este modelo de coordinación se difieren para más adelante, cuando presentemos los sistemas concretos.

Observamos que los espacios de datos compartidos y de comunicación generativa con frecuencia también se consideran como formas de sistemas de publicación y suscripción. En lo que sigue, también adoptaremos este aspecto común. Un buen resumen general de sistemas de publicación y suscripción (desde una perspectiva un tanto amplia) se encuentra en Eugster y colaboradores (2003). En este capítulo consideramos que en estos sistemas existe por lo menos desacoplamiento referencial entre los procesos, pero de preferencia también desacoplamiento temporal.

13.2 ARQUITECTURAS

Un aspecto importante de los sistemas basados en coordinación es que la comunicación ocurre describiendo las características de los elementos de datos que se van a intercambiar. Por consiguiente, la asignación de nombres desempeña un rol crucial. Más adelante en este capítulo regresamos a la asignación de nombres, pero por ahora el tema importante es que, en muchos casos, los elementos de datos no son identificados explícitamente por remitentes y destinatarios.

13.2.1 Enfoque total

Supongamos en primer lugar que los elementos de datos están descritos por una serie de **atributos**. Se dice que un elemento de datos está **publicado** cuando se pone a la disposición de otros procesos para que sea leído. Con esta finalidad, se tiene que transferir una **suscripción** al middleware, la cual describe los elementos de datos en los que el suscriptor está interesado. Tal descripción se compone típicamente de algunos pares (*atributo*, *valor*) posiblemente combinados con pares (*atributo*, *rango*). En el segundo caso, se espera que el atributo especificado adopte valores dentro de un rango especificado. Las descripciones a veces pueden darse utilizando varias clases de predicados formulados sobre los atributos, algo muy similar en naturaleza a las preguntas SQL como en el caso de bases de datos relacionales. Más adelante en este capítulo nos toparemos con estos tipos de descriptores.

Ahora enfrentemos la situación donde las suscripciones tienen que ser **comparadas** con elementos de datos, como se muestra en la figura 13-2. Cuando se da la coincidencia, existen dos posibles escenarios. En el primer caso, el middleware puede decidir remitir los datos publicados a su grupo actual de suscriptores, es decir, a los procesos que tengan una suscripción coincidente. Como alternativa, el middleware también puede remitir una **notificación** al momento en el cual los suscriptores pueden ejecutar una operación read para recuperar el elemento de datos publicado.

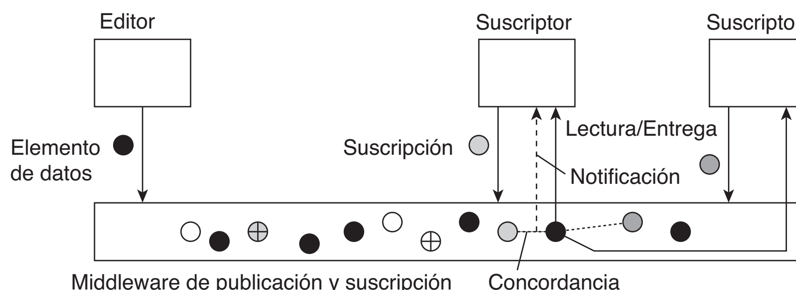


Figura 13-2. Principio de intercambio de elementos de datos entre editores y suscriptores.

En los casos en que los elementos de datos son remitidos de inmediato a los suscriptores, generalmente el middleware no ofrecerá almacenamiento de datos. El almacenamiento es o explícitamente manejado por un servicio aparte, o es responsabilidad de los suscriptores. En otros términos, se tiene un sistema referencialmente desacoplado pero temporalmente acoplado.

Esta situación es diferente cuando se envían notificaciones de modo que los suscriptores tengan que leer explícitamente los datos publicados. Necesariamente, el middleware tendrá que guardar elementos de datos. En estas situaciones, existen operaciones adicionales para implementar la gestión de datos. También es posible anexar un contrato a un elemento de datos de modo que cuando el contrato expire el elemento de datos sea eliminado automáticamente.

En el modelo descrito hasta ahora, se supuso que existe un conjunto fijo de n atributos $a_1, \ldots, a_n$ utilizado para describir elementos de datos. En particular, se supone que cada elemento de

datos publicados tiene un vector asociado <$(a_1, v_1),\ldots,(a_n,v_n)$> de pares (*atributo*, *valor*). En muchos sistemas basados en coordinación, esta suposición es falsa. En cambio, lo que sucede es que los **eventos** son publicados y pueden ser vistos como elementos de datos con sólo un atributo especificado.

Los eventos complican el procesamiento de suscripciones. Como ilustración, consideremos una suscripción tal como "notifica cuando el cuarto R4.20 esté desocupado y la puerta esté abierta". Típicamente, un sistema distribuido que soporta tales suscripciones puede ser implementado colocando sensores independientes para monitorear la ocupación de un cuarto (por ejemplo, sensores de movimiento) y registrar el estado del cerrojo de una puerta. Siguiendo el método descrito hasta ahora, se tendrían que *componer* dichos eventos primitivos en la forma de un elemento de datos publicables a los cuales pueden suscribirse entonces los procesos. La composición de eventos resulta ser una tarea difícil, sobre todo cuando los eventos primitivos son generados por fuentes dispersas a través del sistema distribuido.

Queda claro que, en sistemas basados en coordinación como éstos, el tema crucial es la implementación eficiente y escalable de suscripciones coincidentes para elementos de datos, junto con la construcción de los elementos de datos pertinentes. Desde el exterior, un método de coordinación proporciona un gran potencial para construir sistemas distribuidos a muy grande escala debido al fuerte desacoplamiento de los procesos. Por otra parte, como veremos a continuación, la maquinación de implementaciones escalables sin que se pierda esta independencia no es un ejercicio trivial.

13.2.2 Arquitecturas tradicionales

La solución más simple para igualar elementos de datos a suscripciones es tener una arquitectura cliente-servidor centralizada. Ésta es una solución típica adoptada por muchos sistemas de publicación y suscripción, incluido el WebSphere de IBM (IBM, 2005c) e implementaciones populares para JMS de Sun (Sun Microsystems, 2004a). Asimismo, implementaciones de los modelos de comunicación generativos más elaborados tales como Jini (Sun Microsystems, 2005b) y JavaSpaces (Freeman y cols., 1999) están basados principalmente en servidores centrales. A continuación examinaremos dos ejemplos típicos.

Ejemplo: Jini y JavaSpaces

Jini es un sistema distribuido compuesto a partir de una mezcla de elementos diferentes pero relacionados. Está fuertemente relacionado con el lenguaje de programación Java, aunque muchos de sus principios pueden ser implementados igualmente bien en otros lenguajes. Una parte importante de este sistema está formada por un modelo de coordinación de comunicación generativa. Jini proporciona un desacoplamiento temporal y referencial de procesos mediante un sistema de coordinación llamado **JavaSpaces** (Freeman y cols., 1999), derivado de Linda. Un JavaSpace es un espacio de datos compartido que guarda tuplas que representan un conjunto de referencias a objetos Java. Múltiples JavaSpaces pueden coexistir en un solo sistema Jini.

Las tuplas se guardan en forma serializada. En otros términos, siempre que un proceso desea guardar una tupla, ésta primero se empaqueta, lo cual implica que todos sus campos también se

empaquetan. Por consiguiente, cuando una tupla contiene dos campos diferentes que se refieren al mismo objeto, la tupla guardada en una implementación de JavaSpace mantendrá dos copias empaquetadas de dicho objeto.

Una tupla se pone en JavaSpace por medio de una operación write, la cual primero empaqueta la tupla antes de guardarla. Cada vez que la operación write es invocada en relación con una tupla, se guarda otra copia empaquetada de dicha tupla en el JavaSpace, como se muestra en la figura 13-3. Haremos referencia a cada copia empaquetada como **instancia de tupla**.

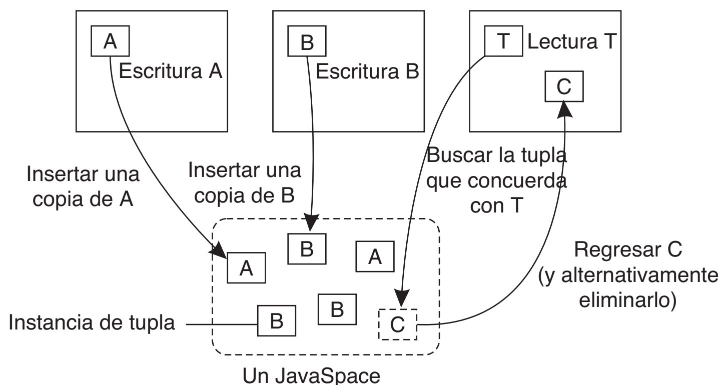


Figura 13-3. Organización general de un JavaSpace en Jini.

El aspecto interesante de la comunicación generativa en Jini es la forma en que las instancias de tupla son leídas desde un JavaSpace. Para leer una instancia de tupla, un proceso proporciona otra tupla que utiliza como **plantilla** para equiparar instancias de tupla guardadas en JavaSpace. Como cualquier otra tupla, una tupla plantilla es un conjunto de referencias a un objeto. Sólo instancias de tupla del mismo tipo que la plantilla pueden ser leídas desde JavaSpace. Un campo en la tupla plantilla o contiene una referencia a un objeto existente o contiene el valor *NULL*. Por ejemplo, consideremos la clase

```
class public Tupla implements Entry {
   public integer id, value;
   public Tupla(Integer id, Integer value){this.id = id; this value = value}
}
```

Entonces una plantilla declarada como

```
Tupla template = new Tupla(null, new Integer(42))
```

equiparará el tupla

```
Tupla item = new Tupla("MyName", new Integer (42))
```

En un JavaSpace, para equiparar una instancia de tupla con un tupla plantilla, éste se empaqueta como siempre, incluidos sus campos *NULL*. Por cada instancia de tupla del mismo tipo que la

plantilla, se hace una comparación campo por campo con la tupla plantilla. Dos campos coinciden si ambos tienen una copia de la misma referencia o si el campo en la tupla plantilla es *NULL*. Una instancia de tupla es igual a una tupla plantilla si sus respectivos campos concuerdan.

Cuando se encuentra que una instancia de tupla coincide con una tupla plantilla provisto como parte de una operación read, dicha instancia de tupla se desempaqueta y regresa al proceso de lectura. También existe una operación take que adicionalmente retira la instancia de tupla del JavaSpace. Ambas operaciones bloquean el invocador hasta que se encuentra una tupla coincidente. También es posible especificar un tiempo máximo de bloqueo. Además, existen variantes que simplemente regresan de inmediato si no existe una tupla coincidente.

Los procesos que utilizan JavaSpaces no tienen que coexistir al mismo tiempo. En realidad, si se implementa un JavaSpace mediante almacenamiento persistente, un sistema Jini completo puede venirse abajo y reiniciarse más tarde sin que se pierda ninguna tupla.

Aunque Jini no lo soporta, deberá quedar claro que tener un servidor central permite que las suscripciones sean bastante elaboradas. Por ejemplo, en el momento en que dos campos nonnull concuerdan si son idénticos. Sin embargo, haciendo que cada campo represente un objeto, la equiparación también podría ser evaluada ejecutando un operador de comparación específico de un objeto [vea también Picco y cols. (2005)]. En realidad, si tal operador puede ser contrarrestado por una aplicación, se puede implementar una semántica de comparación más o menos arbitraria. Es importante señalar que tales comparaciones pueden requerir una extensa búsqueda a través de los elementos de datos actualmente guardados. Tales búsquedas no pueden ser implementadas con facilidad y eficiencia en una forma distribuida. Es exactamente por esta razón que cuando se soportan reglas de equiparación elaboradas, en general, sólo veremos implementaciones centralizadas.

Otra ventaja de tener una implementación centralizada es que se facilita implementar la sincronización de primitivos. Por ejemplo, el hecho de que un proceso pueda bloquearse hasta que se publique un elemento de datos apropiado, y posteriormente se ejecute una lectura destructiva para eliminar la tupla coincidente, ofrece medios para sincronizar el proceso sin que los procesos tengan que conocerse entre sí. De nueva cuenta, en sistemas descentralizados, la sincronización es inherentemente difícil, como también analizamos en el capítulo 6. Más adelante regresaremos a la sincronización.

Ejemplo: TIB/Rendezvous

Una solución alternativa al uso de servidores centrales es diseminar de inmediato los elementos de datos publicados a los suscriptores apropiados mediante multitransmisión. Este principio se utiliza en **TIB/Rendezvous**, cuya arquitectura básica se muestra en la figura 13-4 (TIBCO, 2005). En este método, un elemento de datos es un mensaje etiquetado con una palabra clave compuesta que describe su contenido, tal como *news.comp.os.books*. Un suscriptor proporciona (partes de) una palabra clave o indica los mensajes que desea recibir, tal como *news.comp.*.books*. Se dice que estas palabras clave indican el **tema** de un mensaje.

Para efectuar la implementación de TIB/Rendezvous, es fundamental el uso de transmisión común en redes de área local, aunque también utiliza medios de comunicación más eficientes cuando es posible. Por ejemplo, si se sabe con exactitud dónde reside un suscriptor, en general se

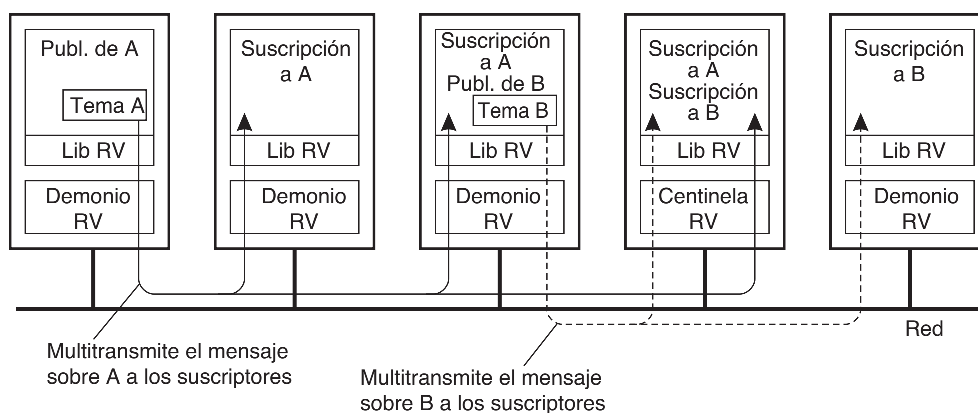


Figura 13-4. Principio de un sistema de publicación y suscripción tal como se implementa en TIB/Rendezvous.

utilizan mensajes de punto a punto. En una red de este tipo, cada servidor ejecutará un **demonio rendezvous**, el cual se encarga de que los mensajes sean enviados y entregados de acuerdo con su tema. Siempre que se publica un mensaje, es multitransmitido a cada servidor en la red que ejecuta un demonio rendezvous. Típicamente, la multitransmisión se implementa usando los medios ofrecidos por la red subyacente, digamos multitransmisión IP o transmisión por medio de hardware.

Los procesos suscritos a un tema transfieren su suscripción a su demonio local. Éste construye una tabla de entradas (*proceso*, *tema*), y siempre que llega un mensaje sobre un tema *S*, el demonio simplemente revisa su tabla en busca de suscriptores locales y remite el mensaje a cada uno. Si no existen suscriptores para *S*, el mensaje se desecha de inmediato.

Cuando se utiliza multitransmisión como se hizo en TIB/Rendezvous, no hay razón por la cual no se puedan elaborar suscripciones que no sean más que una comparación en cadena como actualmente es el caso. La observación crucial aquí es por qué los mensajes son remitidos a cada nodo como sea, la equiparación potencialmente compleja de los datos publicados contra las suscripciones puede hacerse por completo localmente sin más comunicación a través de la red. Sin embargo, como se verá más adelante, son necesarias reglas de comparación simples siempre que se requiera equiparación a través de redes de área amplia.

13.2.3 Arquitecturas punto a punto

Las arquitecturas tradicionales seguidas por la mayoría de los sistemas basados en coordinación sufren de problemas de escalabilidad (aunque sus proveedores comerciales digan lo contrario). Desde luego, tener un servidor central para equiparar las suscripciones con los datos publicados no permite crecer a más de unos cientos de clientes. Asimismo, el uso de multitransmisión requiere medidas especiales para ir más allá del dominio de las redes de área local. Además, si se tiene que

garantizar la escalabilidad, puede que se requieran más restricciones sobre la descripción de suscripciones y datos.

Se ha investigado mucho sobre la realización de sistemas basados en coordinación usando tecnología punto a punto. Existen implementaciones simples para aquellos casos en los que se utilizan palabras clave, ya que éstas pueden ser asignadas a identificadores únicos para datos publicados. Este método también ha sido utilizado para correlacionar pares (*atributo*, *valor*) con identificadores. En estos casos, la equiparación se reduce a una búsqueda simple de un identificador, la cual puede ser implementada eficientemente en un sistema basado en DHT. Este método funciona bien con los sistemas de publicación y suscripción más convencionales, tal como se ilustra en Tam y Jacobsen (2003), pero también para comunicación generativa (Busi y cols., 2004).

Las cosas se complican con esquemas de equiparación más elaborados. Notoriamente difíciles son los casos en que tienen que ser soportados rangos y existen sólo muy pocas propuestas. A continuación, analizamos una de esas propuestas, formulada por uno de los autores y sus colegas (Voulgaris y cols., 2006).

Ejemplo: Sistema de publicación y suscripción basado en conversación

Consideremos un sistema de publicación y suscripción en el cual los elementos de datos pueden ser descritos por medio de N atributos $a_1, \ldots, a_N$ cuyo valor puede ser directamente correlacionado a un número de punto flotante. Tales valores incluyen, por ejemplo, flotantes, enteros, enumeraciones, boleanos, y cadenas. Una suscripción s adopta la forma de una tupla de pares (*atributo*, *valor/rango*), tales como

$$s = <a_1 \rightarrow 3.0, a_4 \rightarrow [0.0, 0.5)>$$

En este ejemplo, s especifica que a_1 deberá ser igual a 3.0, y a_4 deberá quedar en el intervalo [0.0, 0.5). Se permite que otros atributos adopten cualquier valor. Por claridad, supongamos que a cada nodo i entra sólo una suscripción s_i.

Observe que cada suscripción s_i en realidad especifica un subconjunto S_i en el espacio de N–dimensiones de números de punto flotante. A tal subconjunto también se le llama hiperespacio. Para el sistema en conjunto, sólo aquellos datos publicados cuya descripción queda dentro de la unión $\mathbf{S} = \cup S_i$ de estos hiperespacios son de interés. La idea es dividir automáticamente $\mathbf{S}$ en M hiperespacios desunidos $\mathbf{S}_1, \ldots, \mathbf{S}_M$ de modo que cada uno caiga por completo en uno de los hiperespacios de suscripción S_i y juntos abarquen todas las suscripciones. Más formalmente, se tiene que:

$$(\mathbf{S}_m \cup S_i \neq \emptyset) \Rightarrow (\mathbf{S}_m \subseteq S_i)$$

Además, el sistema mantiene a M mínimo en el sentido de que no existe ninguna división con pocas partes $\mathbf{S}_m$. La idea es registrar, por cada hiperespacio $\mathbf{S}_m$, con exactitud los nodos i para los cuales $\mathbf{S}_m \subseteq S_i$. En ese caso, cuando se publica un elemento de datos, el sistema simplemente tiene que buscar el $\mathbf{S}_m$ al cual pertenece el elemento, desde donde puede remitir el elemento a los nodos asociados.

Con este objeto, los nodos regularmente intercambian suscripciones por medio de un protocolo epidémico. Si dos nodos *i* y *j* advierten que sus respectivas suscripciones se entrecruzan, es decir $S_{ij} \equiv S_i \cap S_j \neq \emptyset$ registrarán este hecho y mantendrán las referencias entre ellos. Si descubren un tercer nodo *k* con $S_{ijk} \equiv S_{ij} \cap S_k \neq \emptyset$, los tres se conectarán entre sí de modo que el elemento de datos *d* de S_{ijk} pueda ser diseminado con eficiencia. Observe que si $S_{ij} - S_{ijk} \neq \emptyset$, los nodos *i* y *j* mantendrán sus referencias mutuas, pero ahora asociados con $S_{ij} - S_{ijk}$.

En esencia, lo que se busca es una forma de agrupar nodos en *M* grupos diferentes, de modo que los nodos *i* y *j* pertenezcan al mismo grupo y sólo sus suscripciones S_i y S_j se entrecrucen. Además, los nodos ubicados en el mismo grupo deberán organizarse en una red sobrepuesta que permita la diseminación eficiente de un elemento de datos en el hiperespacio asociado con dicho grupo. Esta situación de un solo atributo se ilustra en la figura 13-5.

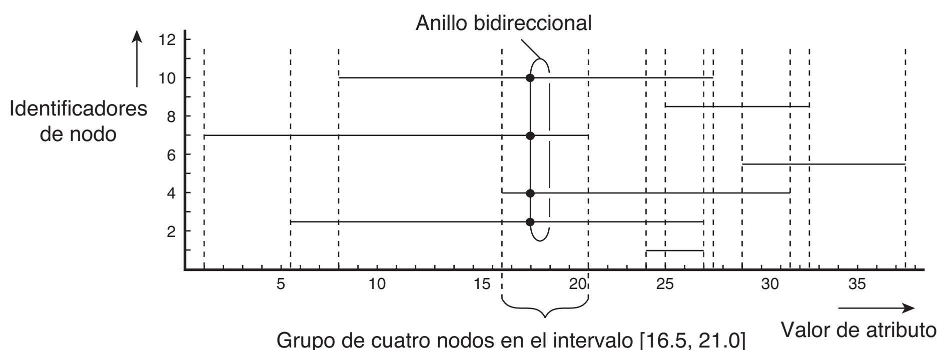


Figura 13-5. Agrupamiento de nodos que soportan consultas de rango en un sistema de publicación y suscripción punto a punto.

Aquí vemos un total de siete nodos donde la línea horizontal del nodo *i* indica su intervalo de interés para el valor del atributo único. También se muestra el agrupamiento de nodos en intervalos de interés desarticulados para valores del atributo. Por ejemplo, los nodos 3, 4, 7 y 10 se agruparán para representar el intervalo [16.5, 21.0]. Cualquier elemento de datos con un valor presente en este intervalo deberá ser diseminado a sólo estos cuatro nodos.

Para construir estos grupos, los nodos se organizan en una red no estructurada basada en conversaciones. Cada nodo mantiene una lista de referencias a otros vecinos (es decir, una **vista parcial**), la cual periódicamente intercambia con uno de sus vecinos como se describe en el capítulo 2. Tal intercambio permitirá que un nodo aprenda sobre otros nodos del sistema. Cada nodo sigue la pista de los nodos que descubre con intereses comunes (es decir, con suscripción entrecruzada).

En cierto momento, cada nodo *i* tendrá generalmente referencias a otros nodos con intereses comunes. Como parte del intercambio de información con el nodo *j*, el nodo *i* ordena estos nodos

por sus identificadores y selecciona el que tenga el identificador más bajo $i_1 > j$, de tal suerte que su suscripción se traslape con la del nodo j, esto es, $S_{j,i_1} \equiv S_{i_1} \cap S_j \neq \emptyset$.

El siguiente a ser seleccionado es $i_2 > i_1$, de modo que su suscripción también se traslape con la de j, pero sólo si contiene elementos aún no cubiertos por el nodo i_1. En otros términos, deberemos tener que $S_{j,i_1,i_2} \equiv (S_{i_2} - S_{j,i_1}) \cap S_j \neq \emptyset$. Este proceso se repite hasta que todos los nodos que tengan un interés traslapado o común con el nodo i hayan sido inspeccionados, lo cual conduce a una lista ordenada $i_1 < i_2 < \cdots < i_n$. Observe que el nodo i_k se encuentra en esta lista porque cubre la región R de interés común de los nodos i y j aún no cubiertos en conjunto por nodos con identificador menor que i_k. En efecto, el nodo i_k es el *primer* nodo al que el nodo j deberá remitir un elemento de datos que se encuentre en esta región única R. Este procedimiento puede ser ampliado para permitir que el nodo i construya un anillo bidireccional. Tal anillo también se muestra en la figura 13-5.

Siempre que se publica un elemento de datos, se disemina tan rápido como es posible hacia *cualquier* nodo interesado en él. Como resultado, con la información disponible en cada nodo encontrar un nodo i interesado en d es simple. De ahí en adelante, el nodo i simplemente tiene que remitir d a lo largo del anillo de suscriptores para el intervalo particular en el que d se encuentra. Para acelerar la diseminación, se mantienen atajos para cada anillo. Los detalles pueden consultarse en Voulgaris y colaboradores (2006).

Análisis

Un método un tanto similar a esta solución basada en conversación en el sentido de que intenta encontrar una partición del espacio cubierto por los valores de atributo, pero el cual utiliza un sistema basado en DHT, se describe en Gupta y colaboradores (2004). En otra propuesta descrita en Bharambe (2004), cada atributo a_i es manejado por un proceso distinto P_i que a su vez particiona el intervalo de su atributo a través de múltiples procesos. Cuando se publica un elemento de datos d, se remite a cada P_i donde posteriormente es guardado en el proceso responsable del valor d de a_i.

Todos estos métodos son ilustrativos por la complejidad cuando se correlaciona un sistema de publicación y sucripción no trivial con una red punto a punto. En esencia, esta complejidad se deriva del hecho de que el soporte de búsquedas en sistemas de asignación de nombres basados en atributos es inherentemente difícil de establecer de un modo descentralizado. Volveremos a enfrentar estas dificultades cuando analicemos la replicación.

13.2.4 Movilidad y coordinación

Un tema que ha recibido considerable atención en la literatura es cómo combinar soluciones de publicación y suscripción con la movilidad de nodos. En muchos casos, se asume que existe una infraestructura básica fija con puntos de acceso para nodos móviles. Conforme a esas suposiciones, el tema se transforma en cómo garantizar que los mensajes publicados no sean entregados más de una vez a un suscriptor que cambia de puntos de acceso. Una solución práctica a este problema es permitir que los suscriptores no pierdan de vista los mensajes que ya recibieron y que simplemente

desechen los duplicados. Soluciones alternativas pero más complicadas comprenden enrutadores o direccionadores que no pierdan de vista qué mensajes fueron enviados a cuáles suscriptores (vea, por ejemplo, Caporuscio y cols., 2003).

Ejemplo: Lime

En el caso de comunicación generativa, se han propuesto varias soluciones para operar un espacio de datos compartido donde (algunos de) los nodos son móviles. Aquí un ejemplo canónico es Lime (Murphy y cols., 2001), el cual se parece muchísimo al modelo JavaSpace que ya analizamos.

En Lime, cada proceso tiene su propio espacio de datos asociado, pero cuando los procesos están próximos entre sí de tal modo que están conectados, sus espacios de datos se vuelven compartidos. Teóricamente, estar conectados puede significar que existe una ruta en una red subyacente conjunta que permite que dos procesos intercambien datos. En la práctica, sin embargo, o significa que dos procesos están temporalmente localizados en el mismo servidor físico o que sus respectivos servidores pueden comunicarse entre sí mediante un enlace inalámbrico (conexión intermedia). Formalmente, los procesos deberán pertenecer al mismo grupo y utilizar el mismo protocolo de comunicación de grupo.

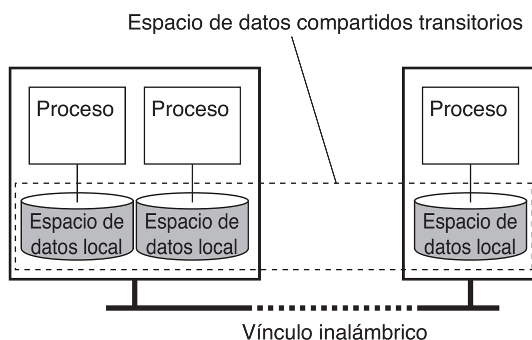


Figura 13-6. Compartimiento transitorio de espacios de datos locales en Lime.

Los espacios de datos locales de procesos conectados forman un espacio de datos transitoriamente compartidos que permiten a los procesos intercambiar tuplas, como se muestra en la figura 13-6. Por ejemplo, cuando un proceso *P* ejecuta una operación write, la tupla asociado se guarda en el espacio de datos local del proceso. En principio, permanece allí hasta que haya una operación take concordante, posiblemente de otro proceso que ahora está en el mismo grupo que *P*. De este modo, el hecho de que en efecto estamos tratando con un espacio de datos compartido completamente distribuido es transparente para los procesos participantes. Sin embargo, Lime también permite romper esta transparencia al especificar con exactitud para quién es una tupla. Asimismo, las operaciones read y take pueden tener un parámetro adicional que especifique de qué proceso se espera una tupla.

Para controlar mejor la distribución de tuplas, los espacios de datos pueden realizar lo que se conoce como **reacciones**. Una reacción especifica una acción que debe ser ejecutada cuando

se encuentra una tupla que concuerda con una plantilla dada en el espacio de datos local. Cada vez que cambia un espacio de datos, se selecciona al azar una reacción ejecutable, que a menudo conduce a una modificación adicional del espacio de datos. Las reacciones cubren el espacio de datos compartidos actual, aunque existen varias restricciones para garantizar que puedan ser ejecutadas con eficiencia. Por ejemplo, en el caso de reacciones débiles, sólo se garantiza que las acciones asociadas son eventualmente ejecutadas siempre que los datos concordantes sigan estando accesibles.

La idea de reacciones se adelanta un paso más en TOTA, donde cada tupla tiene un fragmento de código asociado que señala con exactitud cómo deberá ser movido dicha tupla entre los espacios de datos, posiblemente incluyendo también las transformaciones (Mamei y Zambonelli, 2004).

13.3 PROCESOS

No existe nada realmente especial en cuanto a los procesos utilizados en sistemas de publicación y suscripción. En la mayoría de los casos, se tienen que utilizar mecanismos eficientes para buscar en un conjunto de datos potencialmente grande. El problema principal es elaborar esquemas que funcionen bien en ambientes distribuidos. Más adelante regresamos al tema, cuando analicemos la consistencia y la replicación.

13.4 COMUNICACIÓN

En muchos sistemas de publicación y suscripción, la comunicación es relativamente simple. Por ejemplo, en virtualmente todo sistema basado en Java, toda la comunicación se realiza mediante invocaciones a métodos remotos. Un problema importante que debe ser manejado cuando los sistemas de publicación y suscripción se esparcen a través de un sistema de área amplia es que los datos publicados deberán llegar sólo a los suscriptores pertinentes. Como ya describimos, la utilización de un método autoorganizador mediante el cual se agrupen automáticamente los nodos presentes en un sistema punto a punto, después de lo cual la diseminación ocurra por grupo, es una solución. Una solución alternativa es utilizar enrutamiento basado en el contenido.

13.4.1 Enrutamiento basado en el contenido

En **enrutamiento basado en el contenido**, se supone que el sistema está construido encima de una red punto a punto en la cual los mensajes son explícitamente direccionados entre nodos. En esta configuración es crucial que los enrutadores sean capaces de tomar decisiones de enrutamiento considerando el contenido de un mensaje. Más precisamente, se supone que cada mensaje porta una descripción de su contenido, y que esta descripción puede ser utilizada para interrumpir rutas que no conducen a receptores interesados en dicho mensaje.

Carzaniga y colaboradores (2004) proponen un método práctico para el enrutamiento basado en el contenido. Consideremos un sistema de publicación y suscripción compuesto de *N* servidores

a los que los clientes (es decir, aplicaciones) pueden enviar mensajes, o desde los cuales pueden leerse los mensajes entrantes. Suponemos que para leer mensajes, una aplicación tendrá que haber dado previamente al servidor una descripción de la clase de datos en los que está interesado. El servidor, a su vez, notificará a la aplicación cuando hayan llegado los datos pertinentes.

Carzaniga y colaboradores proponen un esquema de enrutamiento de dos capas donde la capa más baja se compone de un árbol de transmisión compartida que conecta los *N* servidores. Existen varias formas de establecer el árbol y van desde el soporte de multitransmisión a nivel de red hasta árboles de multitransmisión a nivel de aplicación, tal como vimos en el capítulo 4. Aquí, también suponemos que dicho árbol se configuró con los *N* servidores como nodos extremos, junto con un conjunto de nodos intermedios que forman los enrutadores. Observe que la distinción entre un servidor y un enrutador sólo es lógica: una sola máquina puede albergar ambas clases de procesos.

Consideremos primero dos extremos para el enrutamiento basado en el contenido, suponiendo que se tiene que soportar sólo una simple publicación y suscripción basada en el tema donde cada mensaje está etiquetado con una palabra clave única (no compuesta). Una solución extrema es enviar cada mensaje publicado a cada servidor, y dejar que posteriormente éste verifique si cualquiera de sus clientes se había suscrito al tema de dicho mensaje. En esencia, éste es el método seguido en TIB/Rendezvous.

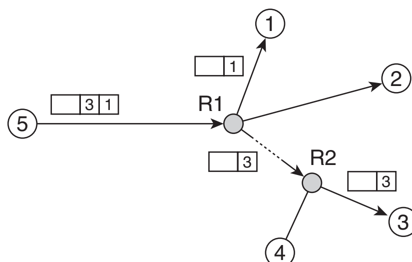


Figura 13-7. Enrutamiento basado en contenido trivial.

La otra solución extrema es permitir que cada servidor transmita sus suscripciones a todos los demás servidores. En consecuencia, cada servidor será capaz de compilar una lista de pares (*tema*, *destino*). Entonces, siempre que una aplicación presente un mensaje sobre el tema *s*, su servidor asociado asigna los servidores destino a dicho mensaje. Cuando el mensaje llega a un enrutador, éste puede utilizar una lista para decidir las rutas que el mensaje deberá seguir, como se muestra en la figura 13-7.

Al tomar esta última aproximación como punto de partida, podemos afinar las capacidades de los enrutadores para decidir adónde remitir los mensajes. Con este objeto, cada servidor transmite su suscripción a través de la red de modo que los enrutadores puedan componer **filtros de enrutamiento**. Por ejemplo, supongamos que el nodo 3 ilustrado en la figura 13-7 se suscribe a los mensajes para los cuales un atributo *a* queda en el intervalo [0,3], pero que el nodo 4 desea mensajes con $a \in [2,5]$. En este caso, el enrutador R_2 creará un filtro de enrutamiento como una tabla con una

entrada por cada uno de sus vínculos de salida (en este caso tres: uno al nodo 3, uno al nodo 4, y uno hacia el enrutador R_1), como se muestra en la figura 13-8.

Interfaz	Filtro
Hacia el nodo 3	$a \in [0,3]$
Hacia el nodo 4	$a \in [2,5]$
Hacia el enrutador R_1	(no especificado)

Figura 13-8. Tabla de enrutamiento parcialmente llena.

Más interesante es lo que sucede en el enrutador R_1. En este ejemplo, las suscripciones de los nodos 3 y 4 dictan que cualquier mensaje con a dentro del intervalo $[0,3] \cup [2,5] = [0,5]$ deberá ser remitido a lo largo de la ruta hacia el enrutador R_2, y ésta es precisamente la información que R_1 guardará en su tabla. No es difícil imaginar qué suscripciones más intrincadas pueden soportarse.

Este ejemplo simple también ilustra que siempre que un nodo abandone el sistema, o cuando ya no esté interesado en mensajes específicos, deberá cancelar su suscripción y esencialmente transmitir esta información a todos los enrutadores. Esta cancelación, a su vez, puede conducir al ajuste de varios filtros de enrutamiento. Ajustes tardíos conducen, en el peor de los casos, a un tráfico innecesario ya que los mensajes pueden ser remitidos a lo largo de rutas para las cuales ya no hay suscriptores. No obstante, se requieren ajustes oportunos para mantener el desempeño a un nivel aceptable.

Uno de los problemas con el enrutamiento basado en el contenido es que aunque el principio de componer filtros de enrutamiento es simple, identificar los vínculos a lo largo de los cuales un mensaje entrante debe ser remitido puede implicar mucho trabajo de cómputo. La complejidad computacional se deriva de la implementación de la equiparación de valores de atributo con suscripciones, lo cual en esencia se reduce a una comparación de entrada por entrada. Cómo se puede realizar con eficiencia esta comparación lo describen Carzaniga y colaboradores (2003).

13.4.2 Soporte de suscripciones compuestas

Los ejemplos dados hasta ahora forman extensiones relativamente simples de las tablas de enrutamiento. Estas extensiones son suficientes cuando las suscripciones adoptan la forma de vectores de pares (*atributo*, *valor/rango*). Sin embargo, a menudo se requieren expresiones de suscripciones más complejas. Por ejemplo, puede ser conveniente expresar **composiciones** de suscripciones en las cuales un proceso especifica en una sola suscripción que está interesado en tipos muy diferentes de elementos de datos. Como ilustración, es posible que un proceso desee ver elementos de datos sobre existencias de IBM *y* datos sobre sus ingresos, pero el envío de elementos de datos de sólo una clase no es útil.

Para manejar composiciones de suscripción, Li y Jacobsen (2005) propusieron diseñar enrutadores análogos a bases de datos de reglas. De hecho, las suscripciones se transforman en reglas que

establecen en qué condiciones los datos publicados deberán ser remitidos, y a lo largo de qué vínculos de salida. No es difícil imaginar que esto puede llevar a esquemas de enrutamiento basados en el contenido mucho más avanzados que los filtros de enrutamiento antes descritos. El soporte de composición de suscripciones está fuertemente relacionado con los temas de asignación de nombres en los sistemas basados en coordinación, los cuales abordamos enseguida.

13.5 ASIGNACIÓN DE NOMBRES

Prestemos ahora un poco más de atención a la asignación de nombres en sistemas basados en coordinación. Hasta aquí, hemos asumido principalmente que todo dato publicado tiene un vector asociado de *n* pares (*atributo, valor*) y que los procesos pueden suscribirse a elementos de datos especificando predicados sobre estos valores de atributo. En general, este esquema de asignación de nombres es fácil de aplicar, aunque los sistemas difieren con respecto a tipos de atributo, valores y los predicados que pueden utilizarse.

Por ejemplo, con JavaSpaces vimos que, en esencia, sólo soporta la comparación para igualdad, aunque ésta puede ser fácilmente ampliada en formas específicas de aplicación. Asimismo, muchos sistemas de publicación y suscripción comerciales soportan sólo operadores de comparación de cadenas un tanto primitivos.

Uno de los problemas ya mencionados es que, en muchos casos, simplemente no se puede asumir que un elemento de datos está etiquetado con valores para todos los atributos. En particular, veremos que un elemento de datos tiene sólo un par asociado (*atributo*, *valor*), en cuyo caso también se conoce como **evento**. El soporte de suscripción a eventos y, en forma notable, a eventos compuestos inspira en gran medida el análisis sobre temas de asignación de nombres en los sistemas de publicación y suscripción. Lo que hemos visto hasta ahora debe ser considerado como los medios más primitivos de soportar coordinación en sistemas distribuidos. A continuación abordaremos más a fondo los eventos y la composición de eventos.

Cuando se trata de eventos compuestos, se tienen que tomar en cuenta dos temas diferentes. El primer tema es describir las composiciones. Tales descripciones constituyen la base para las suscripciones. El segundo tema es cómo recopilar eventos (primitivos) y posteriormente equipararlos con suscripciones. Pietzuch y colaboradores (2003) propusieron una estructura general para la composición de eventos en sistemas distribuidos. Tomamos esta estructura como base de nuestro análisis.

13.5.1 Descripción de eventos compuestos

Primero consideremos algunos ejemplos de eventos compuestos para tener una mejor idea de la complejidad a la cual posiblemente debamos enfrentarnos. La figura 13-9 muestra ejemplos de eventos compuestos cada vez más complejos. En este ejemplo, R4.20 podría ser una sala de computadoras segura y con aire acondicionado.

Las dos primeras suscripciones son relativamente fáciles. S_i es un ejemplo que puede ser manejado por un evento discreto primitivo, en tanto que S_2 es una composición simple de dos eventos discretos. La suscripción S_3 es más compleja ya que requiere que el sistema también sea capaz de

Ejemplo	Descripción
S1	Notifica cuando el cuarto R4.20 está desocupado
S2	Notifica cuando el cuarto R4.20 está desocupado y la puerta está abierta
S3	Notifica cuando el cuarto R4.20 está desocupado durante 10 segundos mientras la puerta está abierta
S4	Notifica cuando la temperatura en el cuarto R4.20 sube más de un grado por cada 30 minutos
S5	Notifica cuando la temperatura promedio en el cuarto R4.20 es de más de 20 grados en los últimos 30 minutos

Figura 13-9. Ejemplos de eventos incluidos en un sistema distribuido.

informar sobre eventos relacionados con el tiempo. Las cosas se complican aún más si las suscripciones implican valores agregados requeridos para computar gradientes (S_4) o promedios (S_5). Observemos que en el caso de S_5 requerimos de un monitoreo continuo del sistema para enviar notificaciones a tiempo.

La idea básica detrás de un lenguaje de composición de eventos para sistemas distribuidos es habilitar la formulación de suscripciones en función de eventos primitivos. En su planteamiento, Pietzuch y colaboradores proporcionan un lenguaje relativamente simple para un tipo ampliado de máquina de estado finito (FSM, por sus siglas en inglés). Las extensiones permiten especificar tiempos de permanencia en estados, así como también la generación de eventos nuevos (compuestos). Los detalles precisos de su lenguaje no son importantes aquí para efectuar nuestro análisis. Lo importante es que las suscripciones pueden ser transformadas en FSMs.

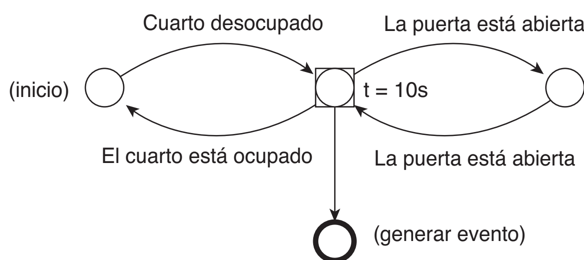


Figura 13-10. Máquina de estado finito para la suscripción S3 de la figura 13-9.

Para dar un ejemplo, la figura 13-10 muestra la FMS para la suscripción S_3 de la figura 13-9. El caso especial está dado por el tiempo medido, indicado mediante la etiqueta "t = 10s" la cual especifica que se hace una transición al estado final si la puerta no se cierra dentro de 10 segundos.

Se pueden describir suscripciones mucho más complejas. Un aspecto importante es que estas FSM a menudo pueden ser descompuestas en FSM más pequeñas que se comunican pasándose eventos entre sí. Observe que esa comunicación de eventos normalmente activaría una transición de estado en la FSM para la cual está pensado dicho evento. Por ejemplo, supongamos que se desea

apagar automáticamente las luces de la habitación R4.20 después de 2 segundos de estar seguros de que ya no hay nadie en ella (y la puerta está cerrada). En ese caso, podemos reutilizar la FSM de la figura 13-10 permitiendo se genere un evento para la segunda FSM que encenderá las luces, como se muestra en la figura 13-11.

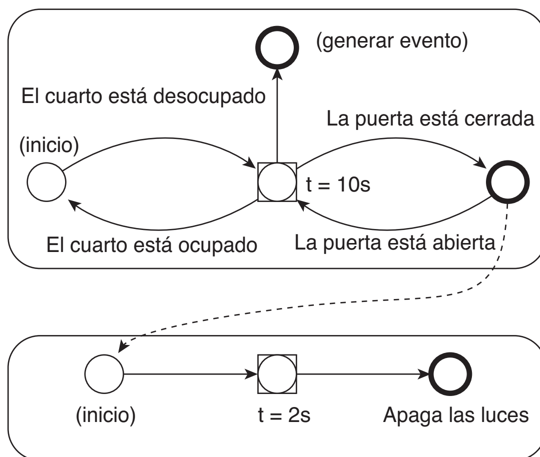


Figura 13-11. Dos máquinas de estado finito acopladas.

La observación importante en este caso es que estas FSM, pueden ser implementadas como procesos aparte en el sistema distribuido. En este caso, la FSM que controla las luces se suscribirá al evento compuesto que se activa cuando la habitación R4.20 está desocupada y la puerta está cerrada. Esto conduce a detectores distribuidos, los cuales analizamos a continuación.

13.5.2 Equiparación de eventos y suscripciones

Consideremos ahora un sistema de publicación y suscripción que soporta eventos compuestos. Cada suscripción se da en la forma de una expresión que puede ser transformada en una máquina de estado finito (FSM). Las transiciones de estado son activadas esencialmente por eventos primitivos que ocurren, tales como salir de un cuarto o cerrar una puerta.

Para equiparar los eventos con las suscripciones, se puede seguir una implementación simple y trivial en la cual cada suscriptor ejecuta un proceso que implementa la máquina de estado finido asociada con su suscripción. En ese caso, todos los eventos primitivos que son pertinentes para una suscripción específica tendrán que ser remitidos al suscriptor. Desde luego, esto en general no será muy eficiente.

Un método mucho mejor es considerar el conjunto completo de suscripciones y descomponerlas en máquinas de estado finito en comunicación, de tal forma que algunas de estas FSM sean compartidas por diferentes suscripciones. Un ejemplo de este compartimiento se muestra en la figura 13-11. Esta aproximación al manejo de suscripciones conduce a lo que se conoce como **detectores de**

eventos distribuidos. Observe que una distribución de detectores de eventos es de naturaleza similar a la resolución distribuida de nombres implementada en varios sistemas de asignación de nombres. Los eventos primitivos conducen a transiciones de estado en máquinas de estado finito relativamente simples, las que a su vez activan la generación de eventos compuestos. Lo segundo puede conducir entonces a transiciones de estado en otras FSM, lo cual de nueva cuenta posiblemente conduzca a más generación de eventos. Desde luego, los eventos se transforman en mensajes enviados a través de la red a procesos suscritos a ellos.

Además de la optimización mediante compartimiento, la descomposición de suscripciones en FSMs comunicantes tiene la ventaja principal de optimizar el uso de la red. Consideremos de nuevo los eventos relacionados con el monitoreo del cuarto de computadoras antes descrito. Suponiendo que existen sólo procesos interesados en los eventos compuestos, es sensato componerlos cerca del cuarto de computadoras. Semejante colocación evitará tener que enviar los eventos primitivos a través de la red. Además, cuando consideramos la figura 13-9, vemos que sólo se puede activar la alarma cuando se advierta que el cuarto ha estado desocupado durante 10 segundos mientras la puerta está abierta. Tal evento generalmente ocurrirá rara vez en comparación con, por ejemplo, abrir (cerrar) la puerta.

La descomposición de suscripciones en detectores de eventos distribuidos, y posteriormente su colocación óptima a través de un sistema distribuido sigue siendo objeto de mucha investigación. Por ejemplo, aún no se ha dicho la última palabra en lenguajes de suscripción, y en especial el compromiso entre expresividad y eficiencia de las implementaciones atraerá mucha atención. En la mayoría de los casos, mientras más expresivo es un lenguaje, menos probabilidades habrá de lograr una implementación distribuida eficiente. Propuestas actuales como las de Demers y colaboradores (2006) y de Liu y Jacobsen (2004) confirman esto. Se requerirán más años antes de que se apliquen estas técnicas a sistemas de publicación y suscripción comerciales.

13.6 SINCRONIZACIÓN

La sincronización en sistemas basados en coordinación, en general, está restringida a sistemas que soportan comunicación generativa. Las cosas son relativamente simples cuando sólo se utiliza un servidor. En ese caso, los procesos simplemente pueden ser bloqueados hasta que las tuplas estén disponibles, pero también es más simple eliminarlos. Las cosas se complican cuando el espacio de datos compartido se replica y distribuye a través de múltiples servidores, como describimos a continuación.

13.7 CONSISTENCIA Y REPLICACIÓN

La replicación desempeña un rol fundamental en la escalabilidad de sistemas basados en coordinación, y principalmente en aquellos de comunicación generativa. A continuación, consideramos primero algunos métodos estándar que ya exploramos en varios sistemas tales como JavaSpaces. Enseguida, describimos algunos resultados recientes que permiten la colocación dinámica y automática de tuplas según sus patrones de acceso.

13.7.1 Métodos estáticos

La implementación distribuida de un sistema que soporta comunicación generativa con frecuencia requiere de atención especial. Ponemos énfasis en las posibles implementaciones de un servidor JavaSpace, es decir, una implementación mediante la cual el conjunto de instancias de tupla pueden ser distribuidas y replicadas en varias máquinas. En Rowstron (2001) encontramos un resumen general de técnicas de implementación de sistemas en tiempo de ejecución basados en tuplas.

Consideraciones generales

Una implementación distribuida eficiente debe resolver dos problemas:

1. Cómo simular el direccionamiento asociativo sin la necesidad de realizar una búsqueda masiva.
2. Cómo distribuir instancias de tupla entre máquinas y localizarlas después.

La clave para ambos problemas es observar que cada tupla es una estructura de datos tecleados. La división del espacio de tupla en subespacios, cada una de cuyas tuplas es del mismo tipo, simplifica la programación y hace posibles ciertas optimizaciones. Por ejemplo, como las tuplas se teclean, se vuelve posible determinar en tiempo de compilación en qué subespacio opera una invocación a write, read o take. Esta división significa que sólo tiene que buscarse una fracción del conjunto de instancias de tupla.

Además, cada subespacio puede ser organizado como una tabla hash utilizando (una parte) de su campo de tupla *i*-ésimo como clave hash. Recuerde que en una instancia de tupla cada campo es una referencia ordenada a un objeto. JavaSpaces no prescribe cómo deberá hacerse el ordenamiento. Por consiguiente, una implementación puede decidir ordenar una referencia de modo que los primeros bytes se utilicen como identificador del tipo del objeto que está siendo ordenado. Una invocación a una operación write, read, o take puede ejecutarse entonces calculando la función hash del campo *i*-ésimo para determinar la posición en la tabla donde pertenece la instancia de tupla. Sabiendo el subespacio y la posición en la tabla se elimina toda la búsqueda. Por supuesto, si el campo *i*-ésimo de una operación read o take es *NULL*, no es posible aplicar la función hash, por lo que generalmente se requiere una búsqueda completa del subespacio. Seleccionando con cuidado el campo donde se aplicará la función hash, sin embargo, a menudo la búsqueda puede ser evitada.

También se utilizan optimizaciones adicionales. Por ejemplo, el esquema de aplicación de funciones hash previamente descrito distribuye las tuplas de un subespacio dado en bandejas para restringir la búsqueda a una sola bandeja. Es posible colocar diferentes bandejas en máquinas diferentes, tanto para repartir la carga más ampliamente como para aprovechar la localidad. Si la función hash es el módulo de identificador de tipo, el número de máquinas y el número de bandejas crecen linealmente con el tamaño del sistema [vea también Bjornson (1993)].

En una red de computadoras, la mejor alternativa depende de la arquitectura de comunicación. Si se dispone de una transmisión confiable, un candidato serio es replicar por completo todos los subespacios en todas las máquinas, como se muestra en la figura 13-12. Cuando se realiza una operación write, la nueva instancia de tupla se transmite e ingresa en el subespacio apropiado en cada máquina. Para realizar una operación read o take, se efectúa una búsqueda en el subespacio local. Sin embargo, como la consumación exitosa de una operación take requiere eliminar la instancia de tupla del JavaSpace, es necesario un protocolo de eliminación para eliminarla de todas las máquinas. Para evitar condiciones apresuradas y detenciones, se puede utilizar un protocolo de realización en dos fases.

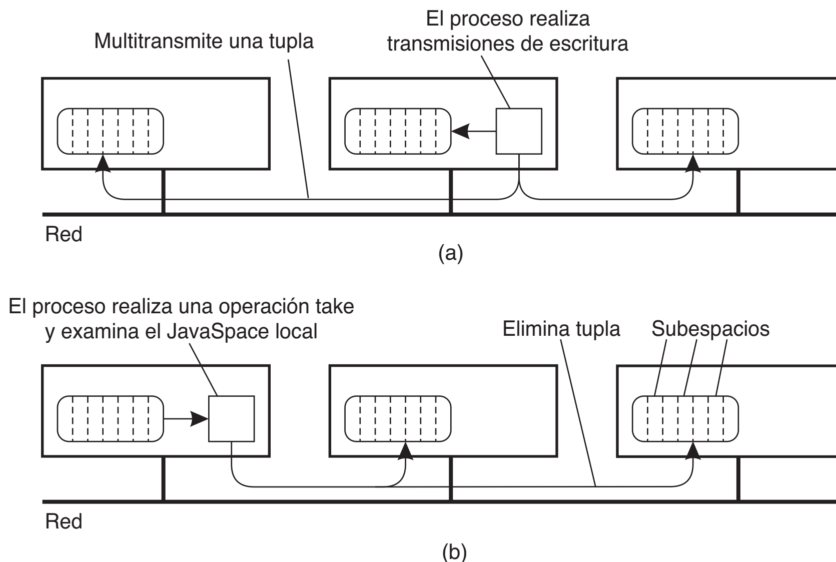


Figura 13-12. Un JavaSpace puede ser replicado en todas las máquinas. Las líneas de puntos muestran la división del JavaSpace en subespacios. (a) Las tuplas se transmiten con la operación write. (b) Las lecturas son locales, pero la eliminación de una instancia cuando se invoca una operación take debe ser transmitida.

Este diseño es simple, pero puede no crecer bien a medida que el sistema crece en el número de instancias de tupla y tamaño de la red. Por ejemplo, implementar este esquema a través de una red de área amplia es prohibitivamente caro.

El diseño inverso es realizar varios write localmente, guardando la instancia de tupla sólo en la máquina que la generó, como se muestra en la figura 13-13. Para realizar una operación read o take, un proceso debe transmitir la tupla plantilla. Cada receptor revisa entonces para ver si tiene uno igual y contesta si lo tiene.

Si la instancia de tupla no está presente, o si no se recibe la transmisión en la máquina que tiene la tupla, la máquina solicitante retransmite la solicitud *ad infinitum*, lo cual incrementa el intervalo

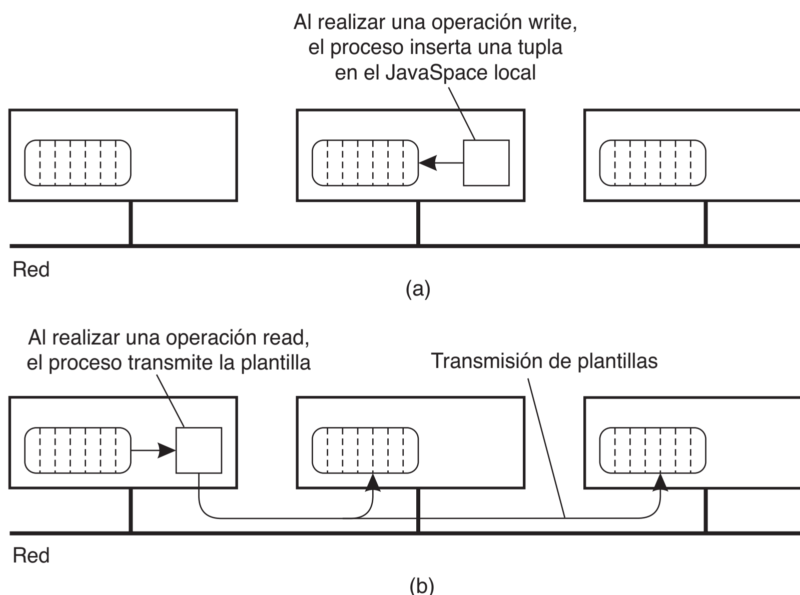


Figura 13-13. JavaSpace no replicado. (a) Se realiza una operación write localmente. (b) Una operación read o take requiere que se transmita la tupla plantilla para localizar una instancia de tupla.

entre transmisiones hasta que una instancia de tupla se materializa y la solicitud puede ser satisfecha. Cuando se envían dos o más instancias de tupla, son tratadas como writes locales y las instancias son efectivamente transferidas desde las máquinas que las tenían hasta la máquina que está haciendo la solicitud. En realidad, el sistema en tiempo de ejecución puede transferir aún más tuplas por su cuenta para equilibrar la carga. Carriero y Gelernter (1986) utilizaron este método para implementar el espacio de tupla Linda en una red de área local.

Estos dos métodos pueden combinarse para producir un sistema con replicación parcial. Como un ejemplo simple, imagínese que todas las máquinas forman lógicamente una cuadrícula rectangular, como se muestra en la figura 13-14. Cuando un proceso localizado en una máquina *A* desea realizar una operación write, transmite (o envía un mensaje punto por punto) la tupla a todas las máquinas ubicadas en su fila de la cuadrícula. Cuando un proceso de la máquina *B* desea leer o tomar una instancia de tupla, transmite la tupla plantilla a todas las máquinas de su columna. Debido a la geometría, siempre habrá exactamente una máquina que ve tanto la instancia de tupla como la tupla plantilla (*C* en este ejemplo), y qué máquina hace la equiparación y envía la instancia de tupla al proceso que la solicitó. Este método es similar al uso de replicación basada en quórum tal como vimos en el capítulo 7.

Las implementaciones analizadas hasta ahora tienen serios problemas de escalabilidad provocados por el hecho de que se requiere multitransmisión o para insertar una tupla en un espacio o para quitar uno. Las implementaciones de área amplia de espacios de tupla no existen. En el mejor de lo casos, varios espacios de tupla *diferentes* pueden coexistir en un solo sistema, donde cada espacio de tupla se implementa en un solo servidor o en una red de área local. Este método se

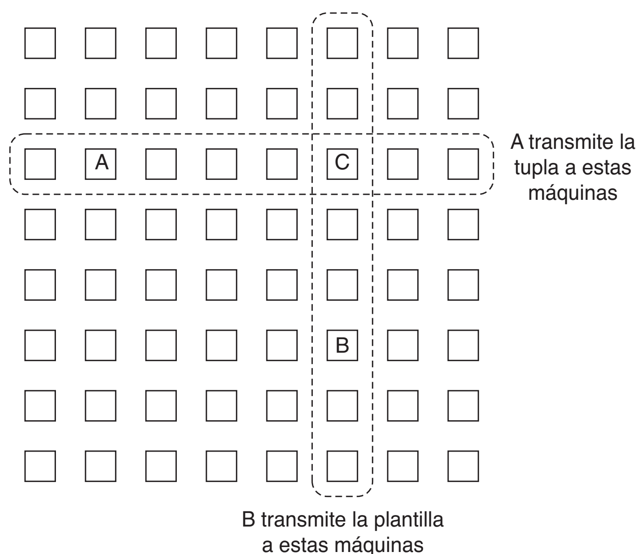


Figura 13-14. Transmisión parcial de tuplas y tuplas plantilla.

utiliza, por ejemplo, en PageSpaces (Ciancarini y cols., 1998) y WCL (Rowstron y Wray, 1998). En WCL, cada servidor de espacio de tupla es responsable de todo un espacio de tupla. En otros términos, un proceso se dirigirá a exactamente un servidor. Sin embargo, es posible migrar un espacio de tupla a un servidor diferente para mejorar el desempeño. Cómo desarrollar una implementación eficiente de espacio de tupla de área amplia sigue siendo una pregunta abierta.

13.7.2 Replicación dinámica

En los sistemas basados en coordinación, en general, la replicación ha estado restringida a políticas estáticas para aplicaciones paralelas como las estudiadas aquí previamente. En aplicaciones comerciales, también se ven esquemas relativamente simples donde espacios de datos completos o, al contrario, partes estáticamente predefinidas de un conjunto de datos se someten a una sola política (GigaSpaces, 2005). Inspirado por la replicación de grano fino de documentos web en Globule, el desempeño también se puede mejorar cuando se diferencia la replicación entre las distintas clases de datos guardados en un espacio de datos. Esta diferenciación es soportada por GigaSpace, el cual examinaremos brevemente en esta sección.

Resumen general de GigaSpace

GigaSpace (GSpace) es un sistema distribuido basado en coordinación y construido encima de JavaSpaces (Rusello y cols., 2004, 2006). En este sistema, la distribución y la replicación de tuplas se realizan por dos razones diferentes: mejorar el desempeño y la disponibilidad. Un elemento clave

en este método es la separación de intereses: las tuplas que deben ser replicadas por disponibilidad puede ser que requieran seguir una estrategia diferente a la de aquellos para los cuales el desempeño está en riesgo. Por esta razón, la arquitectura de GigaSpace soporta varias políticas de replicación de tal forma que diferentes tuplas puedan seguir políticas distintas.

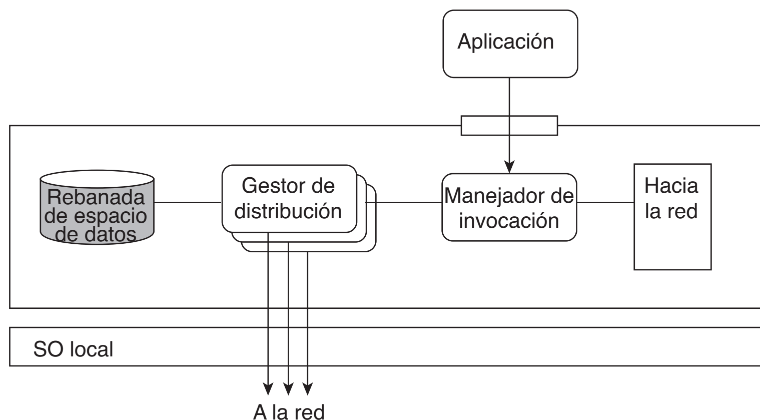


Figura 13-15. Organización interna de un núcleo de GigaSpace.

El funcionamiento principal es relativamente simple. A cada aplicación se le ofrece una interfaz con una interfaz read, write y take, similar a lo que ofrece JavaSpaces. Sin embargo, cada invocación es captada por un manejador de invocaciones local que busca la política a seguirse para la invocación específica. Se selecciona una política basada en el tipo y contenido de la tupla plantilla que se transfiere como parte de la invocación. Cada política es identificada por una plantilla, similar a la forma en que se utilizan plantillas para seleccionar tuplas en otros espacios de datos compartidos basados en Java, como ya vimos.

El resultado de esta selección es una referencia a un gestor de distribución, el cual implementa la misma interfaz, pero ahora de acuerdo con una política de replicación específica. Por ejemplo, si se implementa una política de maestro esclavo, se puede implementar una operación read leyendo de inmediato una tupla del espacio de datos localmente disponible. Asimismo, una operación write puede requerir que el gestor de distribución remita la actualización al nodo maestro y espere un acuse de recibo antes de realizar la operación localmente.

Por último, cada núcleo de GigaSpace dispone de un espacio de datos local, llamado rebanada, la cual se implementa como una versión no distribuida completa de JavaSpaces.

En esta arquitectura (de la cual no se muestran algunos componentes por claridad) se pueden agregar descriptores de política en tiempo de ejecución, y de igual modo también se pueden cambiar los gestores de distribución. Esta configuración permite una sintonización de grano fino de la distribución y replicación de tuplas y, como señalan Russello y colaboradores (2004), dicha sintonización permite un desempeño más alto que es alcanzable con cualquier estrategia global fija aplicada a todas las tuplas en un espacio de datos.

Replicación adaptable

No obstante, el aspecto más importante con sistemas tales como GigaSpace es que la gestión de replicación es automática. En otros términos, en lugar de permitir que el desarrollador de la aplicación decida qué combinación de políticas es la mejor, es preferible permitir que el sistema monitoree los patrones de acceso y el comportamiento para que posteriormente adopte las políticas que se requieran.

Con este objeto, GigaSpace sigue el mismo método que Globule: continuamente mide el ancho de banda de red consumido, la latencia, y el uso de la memoria, y según cuál de estas métricas se considere más importante, coloca tuplas en diferentes nodos y elige la manera más apropiada de mantener las réplicas consistentes. La evaluación de qué política es la mejor para una tupla dada se realiza por medio de un coordinador central que simplemente recopila trazos de los nodos que constituyen el sistema GigaSpace.

Un aspecto interesante es que de vez en cuando necesitamos cambiar de una política de replicación a otra. Existen varias formas en las que tal transición puede ocurrir. Así como GigaSpace intenta separar los mecanismos de las políticas tan bien como sea posible, también puede manejar diferentes **políticas de transición**. El caso por omisión es congelar temporalmente todas las operaciones implementadas para un tipo específico de tupla, eliminar todas las réplicas y reinsertar la tupla en el espacio de datos compartido pero ahora siguiendo la política de replicación recién seleccionada. Sin embargo, según la nueva política de replicación, puede ser posible implementar una forma diferente (y más barata) de hacer la transición. Por ejemplo, cuando se cambia de ninguna replicación a replicación maestro esclavo, un método adecuado podría ser copiar tuplas en los esclavos cuando sean accesados por primera vez.

13.8 TOLERANCIA A FALLAS

Cuando se considera que la tolerancia a fallas es fundamental para cualquier sistema distribuido, resulta un tanto sorpresivo la poca atención que se ha prestado a la tolerancia a fallas en sistemas basados en coordinación, incluidos los de publicación y suscripción básicos, así como los que soportan comunicación generativa. En la mayoría de los casos, la atención se concentra en garantizar una confiabilidad eficiente de la entrega de datos, lo que en esencia se reduce a garantizar una comunicación confiable. Cuando se espera que también el middleware guarde elementos de datos, como es el caso con la comunicación generativa, se pone algo de esfuerzo en busca de un almacenamiento confiable. A continuación examinamos más a fondo estos dos casos.

13.8.1 Comunicación de publicación y suscripción confiables

En sistemas basados en coordinación, donde los elementos de datos publicados se equiparan sólo con suscriptores vivos, la comunicación confiable desempeña un rol crucial. En este caso, la tolerancia a fallas se implementa con más frecuencia habilitando sistemas de multitransmisión confiables

que constituyen el fundamento del software de publicación y suscripción. Existen varios temas que se abordan de una forma general. En primer lugar, independientemente del modo en que ocurra el enrutamiento basado en el contenido, se establece un canal de multitransmisión confiable. En segundo lugar, para manejarlo se requiere un proceso tolerante a fallas. A continuación vemos cómo se abordan estas cuestiones en TIB/Rendezvous.

Ejemplo: Tolerancia a fallas en TIB/Rendezvous

TIB/Rendezvous asume que los medios de comunicación de la red subyacente son inherentemente indignos de confianza. Para compensar esta falta de confianza, siempre que un demonio rendezvous publica un mensaje para otros demonios, lo conservará durante por lo menos 60 segundos. Cuando publica un mensaje, un demonio anexa un número en secuencia (tema independiente) a dicho mensaje. Un demonio receptor puede detectar si le falta un mensaje examinando los números en secuencia (recordemos que los mensajes se entregan a todos los demonios). Cuando falta un mensaje, se pide al demonio editor que lo retransmita.

Esta forma de comunicación confiable no puede evitar que los mensajes se pierdan. Por ejemplo, si un demonio receptor solicita la retransmisión de un mensaje que fue publicado hace más de 60 segundos, el demonio editor generalmente no será capaz de ayudar a recuperar este mensaje perdido. En circunstancias normales, las aplicaciones editora y suscriptora serán notificadas acerca de que ocurrió un error de comunicación. El manejo del error se deja entonces a las aplicaciones.

Mucha de la confiabilidad de comunicación en TIB/Rendezvous está basada en la confiabilidad ofrecida por la red subyacente. TIB/Rendezvous también proporciona multitransmisión confiable utilizando multitransmisión IP (no confiable) como su medio de comunicación subyacente. El esquema seguido en TIB/Rendezvous es un protocolo de multitransmisión a nivel de transporte conocido como **multitransmisión general pragmática** (**PGM**, por sus siglas en inglés), el cual se describe en Speakman y colaboradores (2001). Analicemos brevemente la PGM.

PGM no proporciona fuertes garantías de que al multitransmitir un mensaje éste eventualmente será entregado a cada receptor. La figura 13-16(a) muestra una situación en la que un mensaje ha sido multitransmitido a lo largo de un árbol, pero que no ha sido entregado a dos receptores. PGM confía en que los receptores detectarán que les faltan mensajes por lo que enviarán una solicitud de retransmisión (es decir, un no acuse de recibo) al remitente. Esta solicitud se envía por la ruta inversa en el árbol de multitransmisión enraizado en el remitente, como se muestra en la figura 13-16(b). Siempre que una solicitud de retransmisión llega a un nodo intermedio, tal vez dicho nodo tenga que guardar en caché el mensaje solicitado, situación en la cual manejará la retransmisión. De lo contrario, el nodo simplemente remite el no acuse de recibo al siguiente nodo hacia el remitente. Éste es, en última instancia, el responsable de retransmitir un mensaje.

PGM toma varias medidas para proporcionar una solución escalable a la multitransmisión confiable. En primer lugar, si un nodo intermedio recibe varias solicitudes de retransmisión para exactamente el mismo mensaje, sólo una solicitud de retransmisión es enviada al remitente. De este modo, se intenta garantizar que sólo un no acuse de recibo llegue al remitente para evitar una implosión de retroalimentación. Ya enfrentamos este problema en el capítulo 8, cuando analizamos los temas de escalabilidad en multitransmisión confiable.

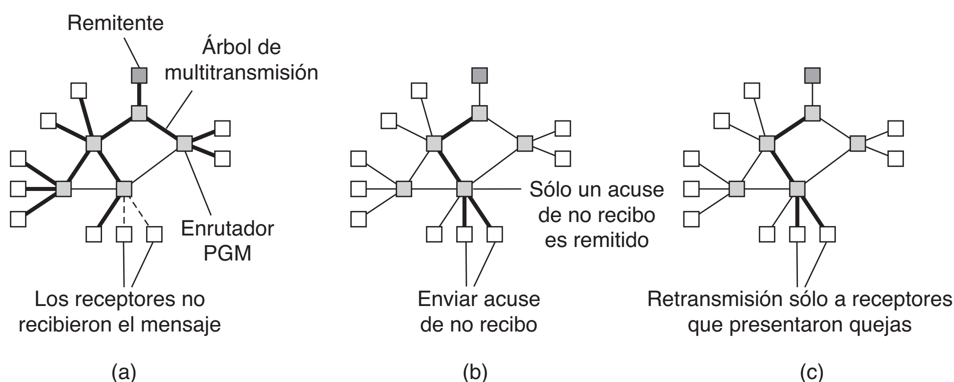


Figura 13-16. Principio de PGM. (a) Se envía un mensaje a lo largo del árbol de multitransmisión. (b) Un enrutador pasará sólo un acuse de no recibo por cada mensaje. (c) Se retransmite un mensaje sólo a los receptores que lo solicitaron.

Una segunda medida tomada por PGM es recordar la ruta que recorre un no acuse de recibo desde los receptores hasta el remitente, como se muestra en la figura 13-16(c). Cuando el remitente por fin retransmite el mensaje solicitado, PGM se encarga de que el mensaje sea multitransmitido sólo a aquellos receptores que habían solicitado la retransmisión. Por consiguiente, los receptores que sí recibieron el mensaje no son molestados por retransmisiones que no les sirven de nada.

Además del esquema de confiabilidad básico y multitransmisión confiable por conducto de PGM, TIB/Rendezvous proporciona más confiabilidad mediante la **entrega de mensajes certificados**. En este caso, un proceso utiliza un canal de comunicación especial para enviar o recibir mensajes. El canal cuenta con un auxiliar asociado, llamado **libro mayor**, para no perder de vista los mensajes certificados enviados y recibidos. Un proceso que desea recibir mensajes certificados se registra con el remitente de tales mensajes. En realidad, el registro permite que el canal maneje más temas de confiabilidad para los cuales los demonios rendezvous no dan soporte. La mayoría de estos temas están ocultos de las aplicaciones y son manejados por la implementación del canal.

Cuando un libro mayor se implementa como archivo, es posible entregar en forma confiable los mensajes incluso cuando falla el proceso. Por ejemplo, cuando un proceso receptor se congela, todos los mensajes que no le llegan hasta que se recupera de nuevo se guardan en el libro mayor del remitente. Una vez que el proceso se recupera, el receptor simplemente se pone en contacto con el libro mayor y le solicita que retransmita los mensajes que no le llegaron.

Para habilitar el ocultamiento de fallas de proceso, TIB/Rendezvous proporciona un medio simple para activar y desactivar automáticamente los procesos. En este contexto, un proceso activo normalmente responde a todos los mensajes que llegan, mientras que un proceso inactivo no lo hace. Un proceso inactivo es un proceso en ejecución que puede manejar sólo eventos especiales, tal como explicaremos en breve.

Los procesos pueden organizarse en un grupo, donde cada proceso tiene un rango único asociado. El rango de un proceso está determinado por su peso (manualmente asignado), pero dos procesos ubicados en el mismo grupo no pueden tener el mismo rango. Para cada grupo, TIB/Rendezvous intentará tener activo cierto número de procesos específicos de grupo, llamado **objetivo activo** del grupo. En muchos casos, el objetivo activo se establece en uno, de modo que toda la comunicación con un grupo se reduzca a un protocolo basado en primario tal como lo vimos en el capítulo 7.

Un proceso activo regularmente envía un mensaje a todos los demás miembros del grupo para anunciar que sigue funcionando. Siempre que falta ese **mensaje de ritmo cardiaco**, el middleware activa en forma automática el proceso de más alto rango que esté entonces inactivo. La activación se logra mediante una devolución de llamada a una operación action, la cual se espera sea implementada por cada miembro de un grupo. Asimismo, cuando un proceso previamente inhibido se recupera y activa, el proceso de más bajo rango entonces activo se desactivará automáticamente.

Para mantenerse consistente con los procesos activos, los procesos inactivos tienen que tomar medidas especiales antes de desactivarse. Un método simple es permitir que un proceso inactivo se suscriba al mismo mensaje que cualquier otro miembro del grupo. Un proceso entrante se procesa como siempre, pero nunca se publican las reacciones. Observe que este esquema es afín con la replicación activa.

13.8.2 Tolerancia a fallas en espacios de datos compartidos

Cuando se trata de comunicación generativa, las cosas se complican. Como también señalan Tolksdorf y Rowstron (2000), en cuanto se tiene que incorporar tolerancia a fallas a espacios de datos compartidos, las soluciones a menudo se vuelven tan ineficientes que sólo son factibles las implementaciones centralizadas. En esos casos, se aplican soluciones tradicionales que utilizan un servidor central respaldado con el uso de un protocolo de respaldo primario simple, combinándolo con la determinación de puntos de control.

Una alternativa es utilizar una replicación más agresiva colocando copias de elementos de datos en diversas máquinas. Este método ha sido adoptado en GigaSpace, esencialmente desplegando los mismos mecanismos que utiliza para mejorar el desempeño mediante replicación. Con este objeto, cada nodo calcula su disponibilidad, la cual emplea luego para calcular la disponibilidad de un solo elemento de datos (replicado) (Russello y cols., 2006).

Para calcular su disponibilidad, un nodo regularmente escribe una marca de tiempo de almacenamiento persistente, ello permite calcular el tiempo en que está funcionando y el tiempo en que no estuvo funcionando. Más precisamente, la disponibilidad se calcula en función del **tiempo medio para la falla** (**MTTF**, por sus siglas en inglés) y del **tiempo medio para la reparación** (**MTTR**, por sus siglas en inglés):

$$Disponibilidad\ de\ nodo = \frac{MTTF}{MTTF + MTTR}$$

Para calcular *MTTF* y *MTTR*, un nodo simplemente ve las marcas de tiempo, como se muestra en la figura 13-17. Esto le permitirá calcular los promedios del tiempo entre fallas, lo cual conduce a una disponibilidad de:

$$\textit{Disponibilidad de nodo} = \frac{\sum_{k=1}^{n} (T_k^{inicio} - T_{k-1}^{terminación})}{\sum_{k=1}^{n} (T_k^{inicio} - T_{k-1}^{terminación}) + \sum_{k=1}^{n} (T_k^{inicio} - T_k^{terminación})}$$

Observe que es necesario registrar con regularidad las marcas de tiempo y que T_k^{inicio} pueda ser tomado sólo como una estimación mejor de cuando ocurrió una congelación. Sin embargo, la disponibilidad así calculada será pesimista, ya que el tiempo real en que un nodo se congela en el tiempo $k^{ésimo}$ será un poco después de T_k^{inicio}. Además, en lugar de tomar promedios desde el principio, también es posible tomar en cuenta solamente las últimas *N* congelaciones.

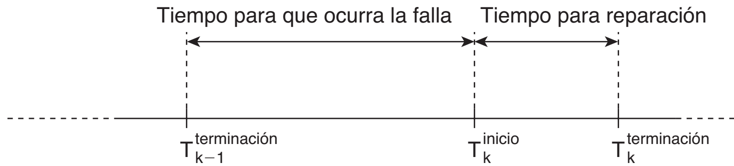


Figura 13-17. Línea del tiempo de un nodo que experimenta fallas.

En GigaSpace, cada tipo de elemento de datos tiene un nodo primario asociado que es responsable de calcular la disponibilidad de ese tipo. Dado que un elemento de datos se replica a través de *m* nodos, su disponibilidad se calcula considerando la disponibilidad a_i de cada uno de los *m* nodos, lo cual nos conduce a:

$$\textit{Disponibilidad de un elemento de datos} = 1 - \prod_{k=1}^{m} (1 - a_i)$$

Si únicamente se toma en cuenta la disponibilidad de un elemento de datos, así como también la de todos los nodos, el primario puede calcular una colocación óptima para un elemento de datos que satisfaga los requerimientos de disponibilidad para un elemento de datos. Además, también se pueden tomar en cuenta otros factores, tales como uso de ancho de banda y cargas de CPU. Advierta que la colocación puede cambiar con el tiempo si estos factores fluctúan.

13.9 SEGURIDAD

En sistemas basados en coordinación, la seguridad plantea un problema difícil. Por un lado, se ha dicho que los procesos deben estar referencialmente desacoplados, pero, por otra parte, se deberá garantizar también la integridad y confidencialidad de los datos. Esta seguridad normalmente se implementa mediante canales seguros (de multitransmisión), lo cuales requieren que efectivamente

los remitentes y receptores puedan autentificarse entre sí. Tal autentificación viola el desacoplamiento referencial.

Para resolver este problema existen diferentes métodos. Un método muy común es configurar una red de agentes que maneje el procesamiento de datos y las suscripciones. Los procesos cliente se pondrán entonces en contacto con los agentes, quienes se encargarán de la autentificación y la autorización. Este tipo de método requiere que los clientes confien en los agentes. Sin embargo, como veremos más adelante, al diferenciar entre tipos de agentes no es necesario que un cliente deba confiar en *todos* los agentes que constituyen el sistema.

Por naturaleza de la coordinación de datos, la autorización normalmente se transforma en temas de confidencialidad. A continuación examinaremos más a fondo estos temas, siguiendo el análisis presentado en Wang y colaboradores (2002).

13.9.1 Confidencialidad

Una diferencia importante entre muchos sistemas distribuidos y los basados en coordinación es que para que sean eficientes, el middleware debe inspeccionar el contenido de los datos publicados. Sin estar habilitado para hacerlo, el middleware esencialmente sólo puede inundar de datos a todos los suscriptores potenciales. Esto plantea el problema de **confidencialidad de la información**, la cual se refiere al hecho de que en ocasiones es importante no permitir que el middleware inspeccione los datos publicados. Este problema se puede evitar por medio de cifrado de extremo a extremo; el sustrato de enrutamiento sólo ve direcciones de origen y destino.

Si los elementos de datos están estructurados en el sentido de que cada elemento contiene múltiples campos, es posible desplegar una secrecía parcial. Por ejemplo, quizá los datos sobre bienes raíces se tengan que enviar entre agentes de una misma oficina con sucursales en diferentes lugares, pero sin revelar la dirección exacta de la propiedad. Para permitir enrutamiento basado en el contenido, el campo de dirección podría ser cifrado, mientras que la descripción de la propiedad podría ser publicada. Con este objeto, Khurana y Koleva (2006) proponen utilizar un esquema de cifrado por cada campo, tal como se presentó en Bertino y Ferrari (2002). En este caso, los agentes que pertenecen a la misma sucursal compartirían la clave secreta para descifrar el campo de dirección. Desde luego, esto viola el desacoplamiento referencial, pero más adelante presentaremos una solución potencial para este problema.

Más problemático es el caso en que ninguno de los campos puede ser revelado al middleware en texto común. La única solución restante es que el enrutamiento basado en el contenido ocurra en los datos cifrados. Como los enrutadores sólo ven datos cifrados, posiblemente por cada campo, las suscripciones tendrán que ser codificadas de tal forma que pueda ocurrir una equiparación parcial. Observe que una equiparación parcial es la base que un enrutador utiliza para decidir por qué vínculo de salida deberá ser remitido el elemento de datos publicados.

Este problema se parece mucho a la búsqueda y consulta de datos cifrados, algo claramente casi imposible de lograr. Como ha sucedido, se sabe que es muy difícil mantener un alto grado de secrecía al mismo tiempo que se ofrece una desempeño razonable (Kantarcioglu y Clifton, 2005).

Uno de los problemas es que si se utiliza cifrado campo por campo, resulta mucho más fácil indagar sobre qué tratan los datos.

Tener que trabajar con datos cifrados también trae a colación el tema de **confidencialidad de las suscripciones**, la cual se refiere al hecho de que las suscripciones tampoco pueden ser reveladas al middleware. En el caso de esquemas de direccionamiento basado en el tema, una solución es utilizar simplemente cifrado por campo y aplicar equiparación sobre una estricta base de campo por campo. La equiparación parcial puede ser acomodada en el caso de palabras clave compuestas, las cuales pueden ser representadas como conjuntos cifrados de sus constituyentes. Un suscriptor enviaría entonces formas cifradas de dichos constituyentes y permitiría que los enrutadores revisen en cuanto a membresía, como también lo sugirieron Raiciu y Rosenblum (2005). Como resulta ser, aún es posible soportar consultas de rango, siempre que se pueda diseñar un esquema eficiente para representar intervalos. Una solución potencial se presenta en Li y colaboradores (2004a).

Por último, la **confidencialidad de la publicación** también es un tema. En este caso, tocamos los mecanismos de control de acceso más tradicionales en los que incluso a ciertos procesos no se les permite ver ciertos mensajes. En esos casos, puede ser que los editores deseen restringir explícitamente el grupo de posibles suscriptores. En muchos casos, este control puede ser ejercido fuera de banda al nivel de las aplicaciones de publicación y suscripción. Sin embargo, puede resultar conveniente que el middleware ofrezca un servicio de manejo de tal control de acceso.

Desacoplamiento de editores y suscriptores

De ser necesario proteger los datos y las suscripciones del middleware, Khurana y Koleva (2006) proponen utilizar un servicio de contabilidad especial (AS), el cual esencialmente se sitúa entre los clientes (editores y suscriptores) y el middleware de publicación y suscripción. La idea básica es desacoplar los editores de los suscriptores al mismo tiempo que se conserva la confidencialidad de la información. En este esquema, los suscriptores registran su interés en elementos de datos específicos, los que posteriormente se enrutan como siempre. Se supone que los elementos de datos contienen campos cifrados. Para permitir el descifrado, una vez que un mensaje debe ser entregado a un suscriptor, el enrutador lo transfiere al servicio de contabilidad donde es *transformado* en un mensaje que sólo el suscriptor puede descifrar. Este esquema se muestra en la figura 13-18.

Un editor se registra en cualquier nodo de la red de publicación y suscripción, es decir, en un agente. Éste remite la información de registro al servicio de contabilidad que luego genera una clave pública que será utilizada por el editor, la cual está firmada por el servicio de contabilidad especial. Desde luego, el servicio de contabilidad especial mantiene la clave privada asociada para sí mismo. Cuando un suscriptor se registra, proporciona una clave de cifrado que es remitida por el agente. Es necesario proseguir a través de una fase de autentificación aparte para garantizar que sólo suscriptores legítimos se registren. Por ejemplo, a los agentes en general no se les permitirá suscribirse para datos publicados.

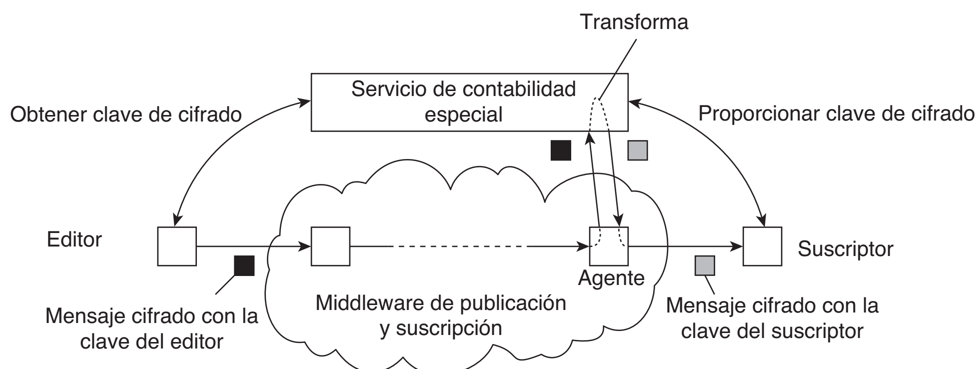


Figura 13-15. Desacoplamiento de los editores desde los suscriptores mediante un servicio confiable adicional.

Ignorando muchos detalles, cuando un elemento de datos es publicado, sus campos críticos tendrán que ser cifrados por el editor. Cuando el elemento de datos llega a un agente que desea transferirlo a un suscriptor, el primero solicita al servicio de contabilidad especial que transforme el mensaje descifrándolo primero y luego cifrándolo con la clave provista por el suscriptor. De este modo, los agentes nunca conocerán el contenido que deberá mantenerse secreto, al mismo tiempo que los editores y suscriptores no tienen que compartir información clave.

Desde luego, es crucial que el servicio de contabilidad especial pueda crecer. Se pueden tomar varias medidas, pero un método razonable es introducir dominios en una forma similar a como lo hace Kerberos. En este caso, puede ser que los mensajes en transmisión tengan que ser transformados volviéndolos a cifrar con la clave pública de un servicio de contabilidad especial ajeno. Para más detalles, el lector debe remitirse a (Khurana y Koleva, 2006).

13.9.2 Espacios de datos compartidos seguros

Se ha realizado muy poco trabajo para hacer que los espacios de datos compartidos sean seguros. Un método común es simplemente cifrar los campos de elementos de datos y permitir que ocurra la equiparación sólo cuando el descifrado tiene éxito y el contenido concuerda con una suscripción. Este método se describe en Vitek y colaboradores (2003). Uno de los problemas importantes con este método es que puede suceder que las claves deban ser compartidas entre editores y suscriptores, o que las claves de descifrado de los editores deban ser conocidas por suscriptores autorizados.

Desde luego, si el espacio de datos compartido es confiable (es decir, si permite que los procesos que lo implementan vean el contenido de las tuplas), las cosas se simplifican. Considerando que la mayoría de las implementaciones utilizan sólo un servidor, la ampliación de dicho servidor con mecanismos de autentificación y autorización a menudo es el método seguido en la práctica.

13.10 RESUMEN

Los sistemas distribuidos basados en coordinación desempeñan un rol importante en la construcción de aplicaciones distribuidas. La mayoría de estos sistemas están enfocados en el desacoplamiento referencial de los procesos, ello significa que éstos no tienen que referirse explícitamente entre sí para habilitar la comunicación. Además, también es posible tener desacoplamiento temporal de modo que los procesos no tengan que coexistir para comunicarse.

Un grupo importante de sistemas basados en coordinación está formado por aquellos sistemas que siguen el paradigma de publicación y suscripción tal como se hizo en TIB/Rendezvous. En este modelo, los mensajes no portan la dirección de su(s) receptor(es), sino que en cambio son direccionados por tema. Los procesos que desean recibir mensajes deberán suscribirse a un tema específico; el middleware se encargará de que los mensajes sean encauzados desde los editores hasta los suscriptores.

Más complejos son los sistemas donde los suscriptores pueden formular predicados sobre los atributos de elementos de datos publicados. En esos casos, se trata de sistemas de publicación y suscripción basados en el contenido. Por eficiencia, es importante que los enrutadores puedan instalar filtros de modo que los datos publicados sean remitidos sólo a través de aquellos vínculos de salida para los que se sepa que hay suscriptores.

Otro grupo de sistemas basados en coordinación utiliza comunicación generativa, la cual ocurre por medio de un espacio de datos compartido de tuplas. Una tupla es una estructura de datos ingresada con el teclado similar a un registro. Para leer una tupla de un espacio de tupla, un proceso especifica lo que está buscando por medio de una tupla plantilla. Se selecciona entonces una tupla que concuerda con la plantilla y se regresa al proceso solicitante. Si no es encontrada una concordancia, el proceso se bloquea.

Los sistemas basados en coordinación son diferentes de muchos otros sistemas distribuidos que se concentran por completo en proporcionar una forma conveniente para que los procesos se comuniquen sin conocerse de antemano. También, la comunicación puede continuar de una forma anónima. La ventaja principal de este método es la flexibilidad, ya que es más fácil ampliar o cambiar un sistema mientras continúa operando.

Los principios de los sistemas distribuidos, tal como fueron analizados en la primera parte del presente libro, se aplican igualmente bien a sistemas basados en coordinación, aunque el almacenamiento en caché y la replicación desempeñan un rol menos prominente en las implementaciones actuales. Además, la asignación de nombres está fuertemente relacionada con la búsqueda basada en atributos soportada por servicios de directorio. El soporte de la seguridad es problemático, ya que esencialmente viola el desacoplamiento entre editores y suscriptores. Los problemas se agravan aún más cuando el middleware debe ser aislado del contenido de los datos publicados, lo cual vuelve más difícil proporcionar soluciones eficientes.

PROBLEMAS

1. ¿En qué tipo de modelo de coordinación clasificaría usted a los sistemas de colas de mensajes estudiados en el capítulo 4?

2. Describa una implementación de un sistema de publicación y suscripción basado en un sistema de una cola de mensajes como el WebSphere de IBM.

3. Explique por qué los sistemas basados en coordinación descentralizados tienen problemas inherentes de escalabilidad.

4. ¿En qué se resuelve un nombre de tema en TIB/Rendezvous y cómo ocurre la resolución del nombre?

5. Describa una implementación simple de la entrega de mensajes totalmente ordenada en un sistema TIB/Rendezvous.

6. En enrutamiento basado en el contenido tal como se utiliza en el sistema Siena, el cual describimos en el texto, se puede enfrentar un serio problema de gestión. ¿Qué problema es ése?

7. Suponga que en un sistema TIB/Rendezvous se replica un proceso. Proporcione dos soluciones para evitar que los mensajes de este proceso replicado se publiquen más de una vez.

8. ¿A qué grado se requiere multitransmisión totalmente ordenada cuando los procesos se replican en un sistema TIB/Rendezvous?

9. Describa un esquema simple para PGM que permita que los receptores detecten pérdida de mensajes, incluso el último mensaje de una serie.

10. ¿Cómo se podría implementar un modelo de coordinación basado en comunicación generativa en TIB/Rendezvous?

11. En Jini siempre se especifica un periodo de contrato como una duración y no como un tiempo absoluto en el cual expira el contrato. ¿Por qué se hace esto?

12. ¿Cuáles son los problemas de escalabilidad más importantes en Jini?

13. Considere una implementación distribuida de un JavaSpace en la cual las tuplas se replican a través de varias máquinas. Proporcione un protocolo para eliminar una tupla de modo que se eviten condiciones apresuradas cuando dos procesos traten de eliminar la misma tupla.

14. Suponga que en Jini una transacción T requiere bloquearse en un objeto que actualmente está bloqueado por otra transacción T'. Explique lo que sucede.

15. Suponga que un cliente Jini guarda en caché la tupla que obtuvo de un JavaSpace de modo que no tenga que ir al JavaSpace la siguiente vez. ¿Tiene sentido este almacenamiento en caché?

16. Responda la pregunta anterior, pero ahora para el caso en que el cliente guarda en caché los resultados regresados por un servicio de búsqueda.

17. Describa una implementación simple de un JavaSpace tolerante a fallas.

18. En algunos sistemas de publicación y suscripción basados en tema, se buscan soluciones seguras en cifrado de extremo a extremo entre editores y suscriptores. Sin embargo, este método puede violar los objetivos de diseño iniciales de los sistemas basados en coordinación. ¿Cómo puede suceder esto?

14

LISTA DE LECTURAS Y BIBLIOGRAFÍA

En los 13 capítulos previos abordamos una diversidad de temas. Este capítulo pretende ayudar a los lectores interesados en ampliar su estudio de los sistemas distribuidos. La sección 14.1 es una lista de lecturas sugeridas. La sección 14.2 es una bibliografía alfabética de todos los libros y artículos citados en este libro.

14.1 SUGERENCIAS PARA LECTURAS ADICIONALES

14.1.1 Introducción y obras generales

Coulouris y colaboradores, *Distributed Systems-Concepts and Design*

Buen texto general sobre sistemas distribuidos. Su cobertura es similar al material incluido en este libro, pero su organización es completamente distinta. Incluye mucho material sobre transacciones distribuidas y algún material más antiguo sobre sistemas de memoria compartida

Foster y Kesselman, T*he Grid 2: Blueprint for a New Computing Infrastructure*

Esta es la segunda edición de un libro en el que muchos expertos en mallas destacan varios temas de computación en malla a gran escala. El libro cubre todos los temas importantes, e incluye muchos ejemplos sobre aplicaciones actuales y futuras.

Neuman, “Scale in Distributed Systems”

Uno de los pocos artículos que proporcionan un resumen general sistemático sobre el tema del escalamiento en sistemas distribuidos. Examina el almacenamiento en caché, la replicación y la distribución como técnicas de escalamiento, además proporciona varias reglas empíricas para aplicar estas técnicas en el diseño de sistemas a gran escala.

Silberschatz y colaboradores, *Applied Operating System Concepts*

Libro de texto general sobre sistemas operativos incluido material sobre sistemas distribuidos con énfasis en sistemas de archivo y coordinación distribuida.

Verissimo y Rodrigues, *Distributed Systems for Systems Architects*

Una avanzada lectura sobre sistemas distribuidos que cubre, básicamente, el mismo material que este libro. Se pone relativamente más énfasis en la tolerancia a fallas y sistemas distribuidos en tiempo real. También se presta atención a la gestión de sistemas distribuidos.

Zhao y Guibas, *Wireless Sensor Networks*

Muchos libros sobre redes sensoras (inalámbricas) describen estos sistemas desde un enfoque de interconexión entre redes. Este libro adopta una perspectiva más de sistemas, por lo que es una lectura atractiva para aquellos interesados en sistemas distribuidos. El libro proporciona una buena cobertura de redes sensoras inalámbricas.

14.1.2 Arquitecturas

Babaoglu y colaboradores, *Self-star Properties in Complex Information Systems*

Se ha dicho mucho sobre sistemas self-*, mas no siempre con el grado de sustancia que se desearía. Este libro contiene varios artículos de autores con una amplia variedad de antecedentes que consideran cómo los aspectos self-* encontraron su camino en los modernos sistemas de computadoras.

Bass y colaboradores, *Software Architecture in Practice*

Este libro, ampliamente utilizado, incluye una excelente introducción práctica y un resumen general de arquitectura de software. Aunque el enfoque no es específicamente hacia sistemas distribuidos, proporciona un excelente fundamento para entender las diversas formas en que pueden ser organizados complejos sistemas de software.

Hellerstein y colaboradores, *Feedback Control of Computing Systems*

Para aquellos lectores con algunos conocimientos matemáticos, este libro proporciona un tratamiento completo sobre cómo se pueden aplicar los bucles de control de retroalimentación a sistemas de computadoras (distribuidos). Como tal, forma una base alternativa para una gran parte de la investigación sobre sistemas de computación autonómicos y self-*.

Lua y colaboradores, "A Survey and Comparison of Peer-to-Peer Overlay Network Schemes"

Excelente estudio de sistemas modernos igual a igual, cubre tanto redes estructuradas como no estructuradas. Este artículo es una buena introducción para aquellos que desean adentrarse más en el tema pero que realmente no saben por dónde empezar.

Oram, *Peer-to-Peer: Harnessing the Power of Disruptive Technologies*

Este libro maneja varios artículos sobre la primera generación de redes igual a igual. Cubre diversos proyectos, así como temas tan importantes como la seguridad, la confianza y la responsabilidad. A pesar de que la tecnología de igual a igual ha registrado un gran progreso, este libro sigue siendo valioso para entender muchos de los temas básicos que deben ser abordados.

White y colaboradores, "An Architectural Approach to Autonomic Computing"

Escrito por el personal técnico que respaldó la idea de computación autonómica, este corto artículo contiene un resumen general de alto nivel de los requerimientos que deben ser satisfechos para implementar sistemas self-*.

14.1.3 Procesos

Andrews, *Foundations of Multithreaded, Parallel, and Distributed Programming*

Si usted siempre necesita una introducción concienzuda a la programación en paralelo y a los sistemas distribuidos, éste es el libro a consultar.

Lewis and Berg, *Multithreaded Programming with Pthreads*

Los Pthreads forman el estándar POSIX para implementar hilos para sistemas operativos y son ampliamente soportados por sistemas basados en UNIX. Aunque los autores se concentran en Pthreads, este libro proporciona una buena introducción a la programación de hilos en general. Como tal, forma una base sólida para el desarrollo de clientes y servidores de hilos múltiples.

Schmidt y colaboradores, *Pattern-Oriented Software Architecture-Patterns for Concurrent and Networked Objects*

Los investigadores también han puesto los ojos en patrones de diseño que son comunes en los sistemas distribuidos. Estos patrones pueden facilitar el desarrollo de sistemas distribuidos ya que permiten a los programadores concentrarse más en temas propios del sistema. En este libro los patrones de diseño analizados comprenden acceso a servicios, manejo de eventos, sincronización y concurrencia.

Smith y Nair, *Virtual Machines: Versatile Platforms for Systems and Processes*

Estos autores también publicaron un breve resumen general sobre virtualización en el ejemplar de mayo de 2005 de *Computer*, pero este libro se adentra en muchos de los detalles (a menudo

intricados) de las máquinas virtuales. Como ya mencionamos en el texto, las máquinas virtuales cada vez adquieren más importancia para los sistemas distribuidos. Este libro es una excelente introducción al tema.

Stevens y Rago, *Advanced Programming in the UNIX Environment*
Si alguna vez se requiere adquirir un solo volumen sobre programación en sistemas UNIX, éste es el libro a considerar. Como cualquier otro libro escrito por el fallecido Richard Stevens, este volumen contiene un caudal de información detallada sobre cómo desarrollar servidores y otros tipos de programas. Esta segunda edición fue ampliada por Rago, que también es muy conocido por escribir libros sobre temas similares.

14.1.4 Comunicación

Birrell y Nelson, "Implementing Remote Procedure Calls"
Un ensayo clásico sobre el diseño y la implementación de uno de los primeros sistemas de llamada a procedimientos remotos.

Hohpe y Woolf, *Enterprise Integration Patterns*
Como cualquier otro material sobre patrones de diseño, este libro proporciona resúmenes generales de alto nivel sobre cómo construir soluciones de mensajería. El libro conforma una excelente lectura para aquellos que desean diseñar soluciones orientadas a mensajes y cubre una gran cantidad de patrones que pueden seguirse durante la fase de diseño.

Peterson y Davie, *Computer Networks, A Systems Approach*
Libro de texto alternativo sobre redes de computadoras que adopta un enfoque un tanto similar a este libro al considerar varios principios y cómo aplicarlos a la interconexión de redes.

Steinmetz y Nahrstedt, *Multimedia Systems*
Buen libro de texto (aunque deficientemente editado) que cubre muchos aspectos de sistemas (distribuidos) de procesamiento multimedia y que, al mismo tiempo, constituye una fina introducción al tema.

14.1.5 Asignación de nombres

Albitz y Liu, *DNS and BIND*
BIND es una implementación públicamente disponible y ampliamente utilizada de un servidor DNS. En este libro, abordamos todos los detalles sobre la configuración de un dominio DNS por medio de BIND. Como tal, proporciona mucha información práctica sobre el más grande servicio de asignación de nombres usado hoy en día.

Balakrishnan y colaboradores, “A Layered Naming Architecture for the Internet”

En este artículo, los autores combinan la asignación de nombres estructurada con la asignación de nombres simple, por lo cual se distinguen tres niveles diferentes: (1) nombres fáciles de usar por humanos los cuales se correlacionan con identificadores de servicio, (2) los identificadores de servicio los cuales se correlacionan con identificadores de punto extremo que identifican en forma única a un host y (3) los puntos extremos correlacionados con direcciones de red. Desde luego, para aquellas partes donde sólo se utilizan identificadores, se puede aplicar convenientemente un sistema basado en DHT.

Balakrishnan y colaboradores, “Looking up Data in P2P Systems”

Buena y de fácil lectura introducción a los mecanismos de búsqueda en sistemas de igual a igual. Se proporcionan sólo unos cuantos detalles sobre el trabajo real de estos mecanismos, pero constituyen un buen punto de partida para abordar una lectura adicional.

Loshin, *Big Book of Lightweight Directory Access Protocol (LDAP) RFC*

Los sistemas basados en LDAP se utilizan mucho en sistemas distribuidos. La fuente final de servicios LDAP son las RFC publicadas por el IETF. Loshin ha recopilado todas las RFC pertinentes en un solo volumen, lo cual lo convierte en una fuente amplia para el diseño y la implementación de servicios LDAP.

Needham, “Names”

Un excelente y fácil de leer artículo sobre el rol de los nombres en sistemas distribuidos. Se pone énfasis en los sistemas de asignación de nombres tal como fueron analizados en la sección 5.3 de este libro, valiéndose de DEC GNS como ejemplo.

Pitoura y Samaras “Locating Objects in Mobile Computing”

Este artículo puede ser utilizado como una amplia introducción a la localización de servicios. Los autores ventilan varias clases de servicios de localización, incluidos aquellos utilizados en sistemas de telecomunicación. El artículo incluye una extensa lista de referencias que puede tomarse como punto de partida para abordar una lectura adicional.

Saltzer, “Naming and Binding Objects”

Aunque se escribió en 1978 y se enfocó en sistemas no distribuidos, este ensayo debería ser el punto de partida de cualquier investigación sobre asignación de nombres. El autor proporciona un excelente tratamiento de la relación entre nombres y objetos, y, en particular, de lo que se requiere para resolver un nombre para un objeto referido. Se presta atención aparte al concepto de mecanismos de cierre.

14.1.6 Sincronización

Guerraoui y Rodrigues, *Introduction to Reliable Distributed Programming*

Título un tanto engañoso para un libro que se concentra en gran medida en algoritmos distribuidos que logran confiabilidad. El libro viene acompañado de un software que permite poner en práctica muchas de las descripciones teóricas.

Lynch, *Distributed Algorithms*

Mediante una sola estructura de soporte, el libro describe muchas clases diferentes de algoritmos distribuidos. Se consideran tres modelos de sincronización: modelos sincrónicos simples, modelos asincrónicos sin suposiciones de sincronización y modelos parcialmente sincrónicos, los cuales se acercan a sistemas reales. Una vez que se acostumbre a la notación teórica, encontrará que este libro contiene muchos algoritmos útiles.

Raynal y Singhal, "Logical Time: Capturing Causality in Distributed Systems"

Este ensayo describe en términos relativamente simples tres tipos de relojes lógicos: tiempo escalar (es decir, marcas de tiempo Lamport), tiempo vectorial y tiempo matricial. Además, describe varias implementaciones que se han utilizado en diversos sistemas distribuidos prácticos y experimentales.

Tel, *Introduction to Distributed Algorithms*

Un libro de texto introductorio alternativo para estudiar algoritmos distribuidos, el cual se concentra únicamente en soluciones para sistemas de transferencia de mensajes. Aunque bastante teórico, en muchos casos el lector puede construir con mucha facilidad soluciones para sistemas reales.

14.1.7 Consistencia y replicación

Adve y Gharachorloo, "Shared Memory Consistency Models: A Tutorial"

Hasta hace poco, ha habido muchos grupos que han desarrollado sistemas distribuidos en los cuales las memorias físicamente dispersas se han juntado en un solo espacio de dirección virtual, lo cual ha llevado a lo que se conoce como sistemas de memoria compartida distribuida. Se han diseñado varios modelos de consistencia de memoria para estos sistemas y forman la base de los modelos analizados en el capítulo 7. Este ensayo proporciona una excelente introducción a estos modelos de consistencia de memoria.

Gray y colaboradores, "The Dangers of Replication and a Solution"

El artículo analiza el compromiso presente entre la replicación que implementa modelos de consistencia secuenciales (llamada replicación ansiosa) y la replicación perezosa. Ambas formas de replicación están formuladas para transacciones. El problema con la replicación ansiosa es su deficiente escalabilidad, en tanto que la replicación perezosa fácilmente puede conducir a resoluciones de conflictos difíciles o imposibles. Los autores proponen un esquema híbrido.

Saito y Shapiro, "Optimistic Replication"

Este artículo presenta una taxonomía de algoritmos de replicación optimistas tal como se utilizan en modelos de consistencia débil. Describe una forma alternativa de considerar la replicación y sus protocolos de consistencia asociados. Un tema interesante en el estudio de escalabilidad de varias soluciones. El artículo también incluye un gran número de referencias útiles.

Sivasubramanian y colaboradores, "Replication for Web Hosting Systems"

En este artículo, los autores examinan muchos aspectos que deben ser abordados para manejar la replicación en sistemas de alojamiento en la web, incluyendo la colocación de réplicas, los protocolos de consistencia y las solicitudes de enrutamiento hacia la mejor réplica. El artículo también incluye una extensa lista del material pertinente.

Wiesmann y colaboradores, "Understanding Replication in Databases and Distributed Systems"

Por tradición, ha habido cierta diferencia entre el manejo de la replicación en bases de datos distribuidas y en sistemas distribuidos de propósito general. En bases de datos, la razón principal para implementar la replicación era mejorar el desempeño. En sistemas distribuidos de propósito general, la replicación se utiliza a menudo para mejorar la tolerancia a las fallas. El artículo presenta una estructura de soporte que permite encontrar soluciones para que estas áreas sean más fáciles de comparar.

14.1.8 Tolerancia a fallas

Birman, *Reliable Distributed Systems*

Escrito por una autoridad en el campo, este libro contiene un caudal de información sobre las trampas de desarrollar sistemas distribuidos altamente confiables. El autor proporciona muchos ejemplos del campo académico y de la industria para ilustrar lo que puede fallar y qué se puede hacer a ese respecto. Cubre una amplia variedad de temas, incluyendo la computación cliente-servidor, los servicios web, los sistemas basados en objetos (CORBA) y también sistemas de igual a igual.

Cristian y Fetzer, "The Timed Asynchronous Distributed System Model"

Este ensayo examina un modelo más realista de sistemas distribuidos aparte de los casos puramente sincrónicos y asincrónicos. Dos suposiciones importantes son que los servicios se complementan dentro de un intervalo de tiempo específico, y que la comunicación es no confiable y está sujeta a fallas de desempeño. El ensayo demuestra la aplicabilidad de este modelo para capturar propiedades importantes de sistemas distribuidos reales.

Guerraoui y Schiper, "Software-Based Replication for Fault Tolerance"

Breve y claro resumen general sobre cómo puede aplicarse la replicación en sistemas distribuidos para mejorar la tolerancia a fallas. Examina la replicación de respaldo primaria y la replicación activa, y relaciona la replicación con la comunicación de grupo.

Jalote, *Fault Tolerance in Distributed Systems*

Uno de los pocos libros de texto dirigidos por completo a la tolerancia de fallas en sistemas distribuidos. El libro cubre la transmisión confiable, la recuperación, la replicación y la elasticidad de los procesos. Existe un capítulo aparte sobre fallas de diseño de software.

Marcus y Stern, *Blueprints for High Availability*

Existen muchas cuestiones a considerar cuando se desarrollan sistemas (distribuidos) para alta disponibilidad. Los autores de este libro adoptan un enfoque pragmático y abordan muchas de las cuestiones técnicas y no técnicas.

14.1.9 Seguridad

Anderson, *Security Engineering: A Guide to Building Dependable Distributed Systems*

Uno de los muy pocos libros encaminado con éxito a la cobertura total del área de seguridad. El libro examina fundamentos tales como contraseñas, control de acceso y criptografía. La seguridad está estrechamente acoplada a dominios de aplicación, y se aborda su implementación en varios dominios: el militar, el bancario, los sistemas médicos, entre otros. Por último, también se examinan aspectos sociales, organizacionales y políticos. Es un gran punto de partida para lectura e investigación adicionales.

Bishop, *Computer Security: Art and Science*

Aunque este libro no está escrito específicamente para sistemas distribuidos, contiene una gran cantidad de información sobre cuestiones generales relacionadas con la seguridad en computadoras, incluidos muchos de los temas estudiados en el capítulo 9. Además, incluye material sobre políticas de seguridad, confianza, evaluación y muchos temas de implementación.

Blaze y colaboradores, "The Role of Trust Management in Distributed Systems Security"

Este artículo argumenta que los sistemas distribuidos a gran escala deberán ser capaces de permitir el acceso a recursos mediante un enfoque más simple que los actuales. En particular, si se sabe que el conjunto de credenciales que acompañan a una petición cumple con una política de seguridad local, la petición debe ser aprobada. En otros términos, la autorización debe ocurrir sin separar la autentificación y el control de acceso. El artículo explica este modelo y muestra cómo puede ser implementado.

Kaufman y colaboradores, *Network Security*

Este libro confiable y frecuentemente ingenioso es el primer lugar donde hay que buscar una introducción a la seguridad en redes. Se explican a fondo los algoritmos y protocolos de claves pública y privada, los hash de mensaje, la autentificación, el sistema Kerberos y el correo electrónico. Las mejores partes son los debates interautores (e incluso intraautores), identificados por subíndices, como en: "Yo$_2$ no podría obtenerme,$_1$ para ser muy específico...".

Menezes y colaboradores, *Handbook of Applied Cryptography*

El título lo dice todo. El libro proporciona el conocimiento matemático necesario para entender las muchas soluciones criptográficas existentes para implementar cifrado, hashing y así sucesivamente. Capítulos completos están dedicados a autentificación, firmas digitales, establecimiento de claves y gestión de claves.

Rafaeli y Hutchison, *A Survey of Key Management for Secure Group Communication*

El título lo dice todo. Los autores abordan varios esquemas que pueden ser utilizados en aquellos sistemas donde grupos de procesos tienen que comunicarse e interactuar en una forma segura. El artículo se concentra en la forma de gestionar y distribuir claves.

Schneier, *Secrets and Lies*

Del mismo autor de *Applied Cryptography*, este libro se enfoca en explicar temas de seguridad para personas no técnicas. Una importante observación es que la seguridad no es sólo una cuestión tecnológica. En realidad, lo que se puede aprender con la lectura de este libro es que tal vez la mayoría de los riesgos relacionados con la seguridad tienen que ver con humanos y con la forma de organización. Como tal, complementa mucho del material presentado en el capítulo 8.

14.1.10 Sistemas basados en objetos distribuidos

Emmerich, *Engineering Distributed Objects*

Excelente libro dedicado por completo a la tecnología de objetos remotos, presta atención específica a CORBA, DCOM y Java RMI. Como tal, proporciona una buena base para comparar estos tres populares modelos de objetos. Además, se presenta material sobre diseño de sistemas valiéndose de objetos remotos, manejo de diferentes formas de comunicación, localización de objetos, persistencia, transacciones y seguridad.

Fleury y Reverbel, “The JBoss Extensible Server”

Muchas aplicaciones web están basadas en el servidor de objetos JBoss J2EE. En este ensayo, los desarrolladores originales del servidor describen los principios subyacentes y el diseño general.

Henning, “The Rise and Fall of CORBA”

Escrito por un experto en el desarrollo de CORBA (quien luego cambio de idea), este artículo contiene fuertes argumentos en contra de CORBA. Más sobresaliente es el hecho de que Henning cree que CORBA es simplemente complejo y que no facilita la vida de los desarrolladores de sistemas distribuidos.

Henning y Vinoski, *Advanced CORBA Programming with C++*

Si usted requiere material sobre programación con CORBA y mientras tanto aprender mucho sobre lo que significa CORBA en la práctica, este libro será su alternativa. Escrito por dos personas involucradas en la especificación y el desarrollo de sistemas CORBA, el libro está lleno de detalles prácticos y técnicos sin limitarse a una implementación específica de CORBA.

14.1.11 Sistemas de archivo distribuidos

Blanco y colaboradores, "A Survey of Data Management in Peer-to-Peer Systems"

Extenso estudio que cubre muchos sistemas de igual a igual. Los autores describen temas de gestión de datos incluyendo la integración de datos, el procesamiento de consultas y la consistencia de datos.

Pate, *UNIX Filesystems: Evolution, Design, and Implementation*

Este libro describe muchos de los sistemas de archivo desarrollados para sistemas UNIX, aunque también contiene un capítulo aparte sobre sistemas de archivo distribuidos. Aporta un resumen general sobre las diversas versiones de NFS, así como sistemas de archivo para grupos de servidores.

Satyanarayanan, "The Evolution of Coda"

Coda es un importante sistema de archivos distribuidos que soporta usuarios móviles. En particular, contiene características avanzadas para soportar lo que se conoce como operaciones desconectadas, mediante las cuales un usuario puede continuar trabajando con su propio conjunto de archivos sin tener que ponerse en contacto con los servidores principales. Este artículo describe cómo ha evolucionado el sistema con los años conforme nuevos requerimientos han salido a la superficie.

Zhu y colaboradores, "Hibernator Helping Disk Arrays Sleep through the Winter"

Los centros de datos utilizan una increíble cantidad de discos para realizar su trabajo. Obviamente, esto requiere una vasta cantidad de energía. Este artículo describe varias técnicas útiles para reducir el consumo de energía; por ejemplo, distinguiendo entre datos muy demandados de los que no lo son tanto.

14.1.12 Sistemas distribuidos basados en la web

Alonso y colaboradores, *Web Services: Concepts, Architectures and Applications*

La popularidad y el embrollo de los servicios web ha provocado un flujo interminable de documentos, tantos que pueden ser catalogados como basura. Por contraste, éste es uno de los muy pocos libros que aportan una descripción clara como el cristal de lo que se tratan los servicios web. Es altamente recomendado como introducción para los novatos, un resumen general para quienes han leído temas intrascendentes, y un ejemplo para aquellos que los producen.

Chappell, *Understanding .NET*

El enfoque que Microsoft ha tomado para soportar el desarrollo de servicios web es combinar muchas de sus técnicas existentes en una sola estructura de soporte junto con la adición de varias funciones nuevas. El resultado se llama .NET. Este enfoque ha provocado mucha confusión en lo referente a qué es en realidad esta estructura de soporte. David Chappel lo explica muy bien.

Fielding, "Principled Design of the Modern Web Architecture"

Del diseñador en jefe del servidor web Apache, este artículo aborda un enfoque general sobre cómo organizar aplicaciones web de tal forma que puedan utilizar mejor el conjunto actual de protocolos.

Podling y Boszormenyi, "A Survey of Web Cache Replacement Strategies"

Apenas si hemos tocado el trabajo que se debe realizar cuando se llenan los cachés web. Este artículo aporta un excelente resumen general de las alternativas que se tienen para desalojar el contenido de los cachés cuando se llenan.

Rabinovich y Spatscheck, *Web Caching and Replication*

Excelente libro que proporciona un resumen general y también muchos detalles sobre distribución de contenido en la web.

Sebesta, *Programming the World Wide Web*

Apenas si se ha tocado el desarrollo de aplicaciones web, lo cual generalmente implica el uso de miles de herramientas y técnicas. Este libro proporciona un amplio resumen general y constituye un buen punto de partida para el desarrollo de sitios web.

14.1.13 Sistemas distribuidos basados en coordinación

Cabri y colaboradores, "Uncoupling Coordination: Tuple-based Models for Mobility"

Los autores proporcionan un buen resumen general de sistemas similares a Linda que pueden operar en ambientes distribuidos móviles. Este artículo también muestra que sigue habiendo mucha investigación en un campo que se inició hace más de 15 años.

Pietzuch y Bacon, "Hermes: A distrib. Event-Based Middleware Architecture"

Hermes es un sistema de publicación y suscripción distribuido desarrollado en University of Cambridge, Reino Unido. Ha sido utilizado como base para efectuar muchos experimentos en sistemas basados en eventos a gran escala, incluida la seguridad. Este artículo describe la organización básica de Hermes.

Wells y colaboradores, "Linda Implementations in Java for Concurrent Systems"

Para aquellos interesados en implementaciones modernas de espacios tuple en Java, este artículo proporciona un buen resumen general. Está más o menos enfocado hacia la computación en lugar de hacia aplicaciones de espacio de tuples generales, no obstante demuestra los diversos compromisos que deben asumirse cuando el desempeño está en riesgo.

Zhao y colaboradores, "Subscription Propagation in Highly-Available Publish/Subscribe Middleware"

Aunque es un artículo un tanto técnico, da una buena idea sobre algunos de los temas que intervienen cuando la disponibilidad es un importante criterio de diseño en sistemas de publicación

y suscripción. En particular, los autores consideran cómo se pueden propagar las actualizaciones cuando las rutas se vuelven redundantes para lograr una alta disponibilidad. No es difícil imaginar que, por ejemplo, la entrega de mensajes sin orden puede ocurrir con facilidad. Tales casos tienen que ser abordados.

14.2 BIBLIOGRAFÍA

ABADI, M. y NEEDHAM, R.: "Prudent Engineering Practice for Cryptographic Protocols." *IEEE Trans. Softw. Eng.* (22)1:6-15, enero de 1996. Citado en la página 400.

ABDULLAHI, S. y RINGWOOD, G.: "Garbage Collecting the Internet: A Survey of Distributed Garbage Collection." *ACM Comput. Surv.* (30)3:330-373, septiembre de 1998. Citado en la página 186.

ABERER, K. y HAUSWIRTH, M.: "Peer-to-Peer Systems." In Singh, M. (ed.), *The Practical Handbook of Internet Computing*, capítulo 35. Boca Raton, FL: CRC Press, 2005. Citado en la página 15.

ABERER, K., ALIMA, L. O., GHODSI, A., GIRDZIJAUSKAS, S., HAUSWIRTH, M. y HARIDI, S.: "The Essence of P2P: A Reference Architecture for Overlay Networks." *Proc. Fifth Int'l Conf. Peer-to-Peer Comput.* (Konstanz, Alemania). Los Alamitos, CA: IEEE Computer Society Press, 2005, págs. 11-20. Citado en la página 44.

ADAR, E. y HUBERMAN, B. A.: "Free Riding on Gnutella." Hewlett Packard, Information Dynamics Lab, enero de 2000. Citado en la página 53.

AIYER, A., ALVISI, L., CLEMENT, A., DAHLIN, M. y MARTIN, J. P.: "BAR Fault Tolerance for Cooperative Services." *Proc. 20th Symp. Operating System Principles* (Brighton, RU). Nueva York, NY: ACM Press, 2005, págs. 45-58. Citado en la página 335.

AKYILDIZ, I. F., SU, W., SANKARASUBRAMANIAM, Y. y CAYIRCI, E.: "A Survey on Sensor Networks." *IEEE Commun. Mag.* (40)8:102-114, agosto de 2002. Citado en la página 28.

AKYILDIZ, I. F., WANG, X. y WANG, W.: "Wireless Mesh Networks: A Survey." *Comp. Netw.* (47)4:445-487, marzo de 2005. Citado en la página 28.

ALBITZ, P. y LIU, C.: *DNS and BIND*. Sebastopol, CA: O'Reilly & Associates, 4a. ed., 2001. Citado en las páginas 210, 560, 626.

ALLEN, R. y LOWE-NORRIS, A.: *Windows 2000 Active Directory*. Sebastopol, CA: O'Reilly & Associates, 2a. ed., 2003. Citado en la página 221.

ALLMAN, M.: "An Evaluation of XML-RPC." *Perf. Eval. Rev.* (30)4:2-11, marzo de 2003. Citado en la página 567.

ALONSO, G., CASATI, F., KUNO, H. y MACHIRAJU, V.: *Web Services: Concepts, Architectures and Applications*. Berlín: Springer-Verlag, 2004. Citado en las páginas 20, 551, 554 y 632.

ALVISI, L. y MARZULLO, K.: "Message Logging: Pessimistic, Optimistic, Causal, and Optimal." *IEEE Trans. Softw. Eng.* (24)2:149-159, febrero de 1998. Citado en las páginas 370 y 371.

AMAR, L., BARAK, A. y SHILOH, A.: "The MOSIX Direct File System Access Method for Supporting Scalable Cluster File Systems." *Cluster Comput.* (7)2:141-150, abril de 2004. Citado en la página 18.

ANDERSON, O. T., LUAN, L., EVERHART, C., PEREIRA, M., SARKAR, R. y XU, J.: "Global Namespace for Files." *IBM Syst. J.* (43)4:702-722, abril de 2004. Citado en la página 512.

ANDERSON, R.: *Security Engineering — A Guide to Building Dependable Distributed Systems*. Nueva York: John Wiley, 2001. Citado en la página 630.

ANDERSON, T., BERSHAD, B., LAZOWSKA, E. y LEVY, H.: "Scheduler Activations: Efficient Kernel Support for the User-Level Management of Parallelism." *Proc. 13th Symp. Operating System Principles*. Nueva York, NY: ACM Press, 1991, págs. 95-109. Citado en la página 75.

ANDREWS, G.: *Foundations of Multithreaded, Parallel, and Distributed Programming*. Reading, MA: Addison-Wesley, 2000. Citado en las páginas 232 y 625.

ANDROUTSELLIS-THEOTOKIS, S. y SPINELLIS, D.: "A Survey of Peer-to-Peer Content Distribution Technologies." *ACM Comput. Surv.* (36)4:335-371, diciembre de 2004. Citado en la página 44.

ARAUJO, F. y RODRIGUES, L.: "Survey on Position-Based Routing." Reporte técnico MINEMA TR-01, University of Lisbon, octubre de 2005. Citado en la página 261.

ARKILLS, B.: *LDAP Directories Explained: An Introduction and Analysis*. Reading, MA: Addison-Wesley, 2003. Citado en la página 218.

ARON, M., SANDERS, D., DRUSCHEL, P. y ZWAENEPOEL, W.: "Scalable Contentaware Request Distribution in Cluster-based Network Servers." *Proc. USENIX Ann. Techn. Conf*. USENIX, 2000, págs. 323-336. Citado en la página 559.

ATTIYA, H. y WELCH, J.: *Distributed Computing Fundamentals, Simulations, and Advanced Topics*. Nueva York: John Wiley, 2a. ed., 2004. Citado en la página 232.

AVIZIENIS, A., LAPRIE, J.-C., RANDELL, B. y LANDWEHR, C.: "Basic Concepts and Taxonomy of Dependable and Secure Computing." *IEEE Trans. Depend. Secure Comput.* (1)1:11-33, enero de 2004. Citado en la página 323.

AWADALLAH, A. y ROSENBLUM, M.: "The vMatrix: A Network of Virtual Machine Monitors for Dynamic Content Distribution." *Proc. Seventh Web Caching Workshop* (Boulder, CO), 2002. Citado en la página 80.

AWADALLAH, A. y ROSENBLUM, M.: "The vMatrix: Server Switching." *Proc. Tenth Workshop on Future Trends in Distributed Computing Systems* (Suzhou, China). Los Alamitos, CA: IEEE Computer Society Press, 2004, págs. 110-118. Citado en la página 94.

BABAOGLU, O., JELASITY, M., MONTRESOR, A., FETZER, C., LEONARDI, S., VAN MOORSEL, A. y VAN STEEN, M. (eds.): *Self-star Properties in Complex Information Systems*, vol. 3460 de *Lect. Notes Comp. Sc*. Berlín: Springer-Verlag, 2005. Citado en las páginas 59 y 624.

BABAOGLU, O. y TOUEG, S.: "Non-Blocking Atomic Commitment." In Mullender, S. (ed.), *Distributed Systems*, págs. 147-168. Wokingham: Addison-Wesley, 2a. ed., 1993. Citado en la página 359.

BABCOCK, B., BABU, S., DATAR, M., MOTWANI, R. y WIDOM, J.: "Models and Issues in Data Stream Systems." *Proc. 21st Symp. on Principles of Distributed Computing* (Monterey, CA). Nueva York, NY: ACM Press, 2002, págs. 1-16. Citado en la página 158.

BAL, H.: *The Shared Data-Object Model as a Paradigm for Programming Distributed Systems*. Tesis de doctorado, Vrije Universiteit, Amsterdam, 1989. Citado en la página 449.

BALAKRISHNAN, H., KAASHOEK, M. F., KARGER, D., MORRIS, R. y STOICA, I.: "Looking up Data in P2P Systems." *Commun. ACM* (46)2:43-48, febrero de 2003. Citado en las páginas 44, 188 y 627.

BALAKRISHNAN, H., LAKSHMINARAYANAN, K., RATNASAMY, S., SHENKER, S., STOICA, I. y WALFISH, M.: "A Layered Naming Architecture for the Internet." *Proc. SIGCOMM* (Portland, OR). Nueva York, NY: ACM Press, 2004, págs. 343-352. Citado en la página 626.

BALAZINSKA, M., BALAKRISHNAN, H. y KARGER, D.: "INS/Twine: A Scalable Peer-to-Peer Architecture for Intentional Resource Discovery." *Proc. First Int'l Conf. Pervasive Computing*, vol. 2414 de *Lect. Notes Comp. Sc.* (Zurich, Suiza), Berlín: Springer-Verlag, 2002, págs. 195-210. Citado en las páginas 222 y 223.

BALLINTIJN, G.: *Locating Objects in a Wide-area System*. Tesis de doctorado, Vrije Universiteit, Amsterdam, 2003. Citado en las páginas 192 y 485.

BARATTO, R. A., NIEH, J. y KIM, L.: "THINC: A Remote Display Architecture for Thin-Client Computing." *Proc. 20th Symp. Operating System Principles* (Brighton, RU). Nueva York, NY: ACM Press, 2005, págs. 277-290. Citado en las páginas 85 y 86.

BARBORAK, M., MALEK, M. y DAHBURA, A.: "The Consensus Problem in Fault-Tolerant Computing." *ACM Comput. Surv.* (25)2:171-220, junio de 1993. Citado en la página 335.

BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAR, R., PRATT, I. y WARFIELD, A.: "Xen and the Art of Virtualization." *Proc. 19th Symp. Operating System Principles* (Bolton Landing, NY). Nueva York, NY: ACM Press, 2003, págs. 164-177. Citado en la página 81.

BARKER, W.: "Recommendation for the Triple Data Encryption Algorithm (TDEA) Block Cipher. "NIST Special Publication 800-67, mayo de 2004. Citado en la página 393.

BARRON, D.: *Pascal — The Language and its Implementation*. Nueva York: John Wiley, 1981. Citado en la página 110.

BARROSO, L., DEAM, J. y HOLZE, U.: "Web Search for a Planet: The Google Cluster Architecture." *IEEE Micro* (23)2:21-28, marzo de 2003. Citado en la página 497.

BARYSHNIKOV, Y., COFFMAN, E. G., PIERRE, G., RUBENSTEIN, D., SQUILLANTE, M. y YIMWADSANA, T.: "Predictability of Web-Server Traffic Congestion." *Proc. Tenth Web Caching Workshop,* (Sophia Antipolis, Francia). IEEE, 2005, págs. 97-103. Citado en la página 576.

BASILE, C., KALBARCZYK, Z. y IYER, R. K.: "A Preemptive Deterministic Scheduling Algorithm for Multithreaded Replicas." *Proc. Int'l Conf. Dependable Systems and Networks* (San Francisco, CA) Los Alamitos, CA: IEEE Computer Society Press, 2003, págs. 149-158. Citado en la página 474.

BASILE, C., WHISNANT, K., KALBARCZYK, Z. y IYER, R. K.: "Loose Synchronization of Multithreaded Replicas." *Proc. 21st Symp. on Reliable Distributed Systems* (Osaka, Japón). Los Alamitos, CA: IEEE Computer Society Press, 2002, págs. 250-255. Citado en la página 474.

BASS, L., CLEMENTS, P. y KAZMAN, R.: *Software Architecture in Practice*. Reading, MA: Addison-Wesley, 2a. ed., 2003. Citado en las páginas 34, 35, 36 y 624.

BAVIER, A., BOWMAN, M., CHUN, B., CULLER, D., KARLIN, S., MUIR, S., PETERSON, L., ROSCOE, T., SPALINK, T. y WAWRZONIAK, M.: "Operating System Support for Planetary-Scale Network Services." *Proc. First Symp. Networked Systems Design and Impl.* (San Francisco, CA). Berkeley, CA: USENIX, 2004, págs. 245-266. Citado en las páginas 99 y 102.

BERNERS-LEE, T., CAILLIAU, R., NIELSON, H. F., y SECRET, A.: "The World-Wide Web." *Commun. ACM* (37)8:76-82, agosto de 1994. Citado en la página 545.

BERNERS-LEE, T., FIELDING, R., y MASINTER, L.: "Uniform Resource Identifiers (URI): Generic Syntax." RFC 3986, enero de 2005. Citado en la página 567.

BERNSTEIN, P.: "Middleware: A Model for Distributed System Services." *Commun. ACM* (39)2:87-98, febrero de 1996. Citado en la página 20.

BERNSTEIN, P., HADZILACOS, V., y GOODMAN, N.: *Concurrency Control and Recovery in Database Systems*. Reading, MA: Addison-Wesley, 1987. Citado en las páginas 355 y 363.

BERSHAD, B., ZEKAUSKAS, M. y SAWDON, W.: "The Midway Distributed Shared Memory System." *Proc. COMPCON*. IEEE, 1993, págs. 528-537. Citado en la página 286.

BERTINO, E. y FERRARI, E.: "Secure and Selective Dissemination of XML Documents." *ACM Trans. Inf. Syst. Sec.* (5)3:290-331, 2002. Citado en la página 618.

BHAGWAN, R., TATI, K., CHENG, Y., SAVAGE, S. y VOELKER, G. M.: "Total Recall: Systems Support for Automated Availability Management." *Proc. First Symp. Networked Systems Design and Impl.* (San Francisco, CA). Berkeley, CA: USENIX, 2004, págs. 337-350. Citado en la página 532.

BHARAMBE, A. R., AGRAWAL, M. y SESHAN, S.: "Mercury: Supporting Scalable Multi-Attribute Range Queries". *Proc. SIGCOMM* (Portland, OR). Nueva York, NY: ACM Press, 2004, págs. 353-366. Citado en las páginas 225 y 599.

BIRMAN, K.: *Reliable Distributed Systems: Technologies, Web Services, and Applications*. Berlín: Springer-Verlag, 2005. Citado en las páginas 90, 335, 582 y 629.

BIRMAN, K.: "A Response to Cheriton and Skeen's Criticism of Causal and Totally Ordered Communication." *Oper. Syst. Rev.* (28)1:11-21, enero de 1994. Citado en la página 251.

BIRMAN, K. y JOSEPH, T.: "Reliable Communication in the Presence of Failures." *ACM Trans. Comp. Syst.* (5)1:47-76, febrero de 1987. Citado en la página 350.

BIRMAN, K., SCHIPER, A. y STEPHENSON, P.: "Lightweight Causal and Atomic Group Multicast." *ACM Trans. Comp. Syst.* (9)3:272-314, agosto de 1991. Citado en la página 353.

BIRMAN, K. y VAN RENESSE, R. (eds.): *Reliable Distributed Computing with the Isis Toolkit*. Los Alamitos, CA: IEEE Computer Society Press, 1994. Citado en la página 251.

BIRRELL, A. y NELSON, B.: "Implementing Remote Procedure Calls." *ACM Trans. Comp. Syst.* (2)1:39-59, febrero de 1984. Citado en las páginas 126 y 626.

BISHOP, M.: *Computer Security: Art and Science*. Reading, MA: Addison-Wesley, 2003. Citado en las páginas 385 y 630.

BJORNSON, R.: *Linda on Distributed Memory Multicomputers*. Tesis de doctorado, Yale University, Department of Computer Science, 1993. Citado en la página 608.

BLACK, A. y ARTSY, Y.: "Implementing Location Independent Invocation." *IEEE Trans. Par. Distr. Syst.* (1)1:107-119, enero de 1990. Citado en la página 186.

BLAIR, G., COULSON, G. y GRACE, P.: "Research Directions in Reflective Middleware: the Lancaster Experience." *Proc. Third Workshop Reflective & Adaptive Middleware* (Toronto, Canadá). Nueva York, NY: ACM Press, 2004, págs. 262-267. Citado en la página 58.

BLAIR, G. y STEFANI, J.-B.: *Open Distributed Processing and Multimedia*. Reading, MA: Addison-Wesley, 1998. Citado en las páginas 8 y 165.

BLAKE-WILSON, S., NYSTROM, M., HOPWOOD, D., MIKKELSEN, J. y WRIGHT, T.: "Transport Layer Security (TLS) Extensions." RFC 3546, junio de 2003. Citado en la página 584.

BLANCO, R., AHMED, N., HADALLER, D., SUNG, L. G. A., LI, H. y SOLIMAN, M. A.: "A Survey of Data Management in Peer-to-Peer Systems." Reporte técnico CS-2006-18, University of Waterloo, Canadá, junio de 2006. Citado en la página 632.

BLAZE, M., FEIGENBAUM, J., IOANNIDIS, J. y KEROMYTIS, A.: "The Role of Trust Management in Distributed Systems Security." En Vitek, J. y Jensen, C. (eds.), *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, vol. 1603 de *Lect. Notes Comp. Sc.*, págs. 185-210. Berlín: Springer-Verlag, 1999. Citado en la página 630.

BLAZE, M.: *Caching in Large-Scale Distributed File Systems*. Tesis de doctorado, Department of Computer Science, Princeton University, enero de 1993. Citado en la página 301.

BONNET, P., GEHRKE, J. y SESHADRI, P.: "Towards Sensor Database Systems." *Proc. Second Int'l Conf. Mobile Data Mgt.*, vol. 1987 de *Lect. Notes Comp. Sc.* (Hong Kong, China). Berlín: Springer-Verlag, 2002, págs. 3-14. Citado en la página 29.

BOOTH, D., HAAS, H., MCCABE, F., NEWCOMER, E., CHAMPION, M., FERRIS, C. y ORCHARD, D.: "Web Services Architecture". W3C Working Group Note, febrero de 2004. Citado en la página 551.

BOUCHENAK, S., BOYER, F., HAGIMONT, D., KRAKOWIAK, S., MOS, A., DE PALMA3, N., QUEMA3, V. y STEFANI, J.-B.: "Architecture-Based Autonomous Repair Management: An Application to J2EE Clusters." *Proc. 24th Symp. on Reliable Distributed Systems* (Orlando, FL). Los Alamitos, CA: IEEE Computer Society Press, 2005, págs. 13-24. Citado en la página 65.

BREWER, E.: "Lessons from Giant-Scale Services." *IEEE Internet Comput.* (5)4:46-55, julio de 2001. Citado en la página 98.

BRUNETON, E., COUPAYE, T., LECLERCQ, M., QUEMA, V. y STEFANI, J.-B.: "An Open Component Model and Its Support in Java." *Proc. Seventh Int'l Symp. Component based Softw. Eng.*, vol. 3054 de *Lect. Notes Comp. Sc.* (Edinburgh, RU). Berlín: Springer-Verlag, 2004, págs. 7-22. Citado en la página 65.

BUDHIJARA, N., MARZULLO, K., SCHNEIDER, F. y TOUEG, S.: "The Primary-Backup Approach." En Mullender, S. (ed.), *Distributed Systems*, págs. 199-216. Wokingham: Addison-Wesley, 2a. ed., 1993. Citado en la página 308.

BUDHIRAJA, N. y MARZULLO, K.: "Tradeoffs in Implementing Primary-Backup Protocols. "Reporte técnico TR 92-1307, Department of Computer Science, Cornell University, 1992. Citado en la página 309.

BURNS, R. C., REES, R. M., STOCKMEYER, L. J. y LONG, D. D. E.: "Scalable Session Locking for a Distributed File System." *Cluster Computing* (4)4:295-306, octubre de 2001. Citado en la página 518.

BUSI, N., MONTRESOR, A. y ZAVATTARO, G.: "Data-driven Coordination in Peer-to-Peer Information Systems." *Int'l J. Coop. Inf. Syst.* (13)1:63-89, marzo de 2004. Citado en la página 597.

BUTT, A. R., JOHNSON, T. A., ZHENG, Y. y HU, Y. C.: "Kosha: A Peer-to-Peer Enhancement for the Network File System." *Proc. Int'l Conf. Supercomputing* (Washington, DC). Los Alamitos, CA: IEEE Computer Society Press, 2004, págs. 51-61. Citado en la página 500.

CABRI, G., FERRARI, L., LEONARDI, L., MAMEI, M. y ZAMBONELLI, F.: "Uncoupling Coordination: Tuple-based Models for Mobility." En Paolo Bellavista y Antonio Corradi (eds.), *The Handbook of Mobile Middleware*. Londres, RU: CRC Press, 2006. Citado en la página 633.

CABRI, G., LEONARDI, L., y ZAMBONELLI, F.: "Mobile-Agent Coordination Models for Internet Applications." *IEEE Computer* (33)2:82-89, febrero de 2000. Citado en la página 590.

CAI, M., CHERVENAK, A. y FRANK, M.: "A Peer-to-Peer Replica Location Service Based on A Distributed Hash Table." *Proc. High Perf. Comput., Netw., & Storage Conf.* (Pittsburgh, PA). Nueva York, NY: ACM Press, 2004, págs. 56-67. Citado en la página 529.

CALLAGHAN, B.: *NFS Illustrated*. Reading, MA: Addison-Wesley, 2000. Citado en las páginas 492 y 510.

CANDEA, G., BROWN, A. B., FOX, A. y PATTERSON, D.: "Recovery-Oriented Computing: Building Multitier Dependability." *IEEE Computer* (37)11:60-67, noviembre de 2004a. Citado en la página 372.

CANDEA, G., KAWAMOTO, S., FUJIKI, Y., FRIEDMAN, G. y FOX, A.: "Microreboot: A Technique for Cheap Recovery." *Proc. Sixth Symp. on Operating System Design and Implementation* (San Francisco, CA). Berkeley, CA: USENIX, 2004b, págs. 31-44. Citado en la página 372.

CANDEA, G., KICIMAN, E., KAWAMOTO, S. y FOX, A.: "Autonomous Recovery in Componentized Internet Applications." *Cluster Comput.* (9)2:175-190, febrero de 2006. Citado en la página 372.

CANTIN, J., LIPASTI, M. y SMITH, J.: "The Complexity of Verifying Memory Coherence and Consistency." *IEEE Trans. Par. Distr. Syst.* (16)7:663-671, julio de 2005. Citado en la página 288.

CAO, L. y OSZU, T.: "Evaluation of Strong Consistency Web Caching Techniques". *World Wide Web* (5)2:95-123, junio de 2002. Citado en la página 573.

CAO, P. y LIU, C.: "Maintaining Strong Cache Consistency in the World Wide Web." *IEEE Trans. Comp.* (47)4:445-457, abril de 1998. Citado en la página 573.

CAPORUSCIO, M., CARZANIGA, A. y WOLF, A. L.: "Design and Evaluation of a Support Service for Mobile, Wireless Publish/Subscribe Applications." *IEEE Trans. Softw. Eng.* (29)12:1059-1071, diciembre de 2003. Citado en la página 600.

CARDELLINI, V., CASALICCHIO, E., COLAJANNI, M. y YU, P.: "The State of the Art in Locally Distributed Web-Server Systems." *ACM Comput. Surv.* (34)2:263-311, junio de 2002. Citado en la página 560.

CARRIERO, N. y GELERNTER, D.: "The S/Net's Linda Kernel." *ACM Trans. Comp. Syst.* (32)2:110-129, mayo de 1986. Citado en la página 609.

CARZANIGA, A., RUTHERFORD, M. J. y WOLF, A. L.: "A Routing Scheme for Content-Based Networking." *Proc. 23rd INFOCOM Conf.*, (Hong Kong, China). Los Alamitos, CA: IEEE Computer Society Press, 2004. Citado en la página 601.

CARZANIGA, A. y WOLF, A. L.: "Forwarding in a Content-based Network." *Proc. SIGCOMM* (Karlsruhe, Alemania). Nueva York, NY: ACM Press, 2003, págs. 163-174. Citado en la página 603.

CASTRO, M., DRUSCHEL, P., GANESH, A., ROWSTRON, A. y WALLACH, D. S.: "Secure Routing for Structured Peer-to-Peer Overlay Networks." *Proc. Fifth Symp. on Operating System Design and Implementation* (Boston, MA). Nueva York, NY: ACM Press, 2002a, págs. 299-314. Citado en las páginas 539 y 540.

CASTRO, M., DRUSCHEL, P., HU, Y. C. y ROWSTRON, A.: "Topology-aware Routing in Structured Peer-to-Peer Overlay Networks." Reporte técnico MSR-TR-2002-82, Microsoft Research, Cambridge, RU, junio de 2002b. Citado en la página 190.

CASTRO, M., RODRIGUES, R. y LISKOV, B.: "BASE: Using Abstraction to Improve Fault Tolerance." *ACM Trans. Comp. Syst.* (21)3:236-269, agosto de 2003. Citado en la página 531.

CASTRO, M., COSTA, M. y ROWSTRON, A.: "Debunking Some Myths about Structured and Unstructured Overlays." *Proc. Second Symp. Networked Systems Design and Impl.* (Boston, MA). Berkeley, CA: USENIX, 2005. Citado en la página 49.

CASTRO, M., DRUSCHEL, P., KERMARREC, A.-M. y ROWSTRON, A.: "Scribe: A Large-Scale and Decentralized Application-Level Multicast Infrastructure." *IEEE J. Selected Areas Commun.* (20)8:100-110, octubre de 2002. Citado en la página 167.

CASTRO, M. y LISKOV, B.: "Practical Byzantine Fault Tolerance and Proactive Recovery." *ACM Trans. Comp. Syst.* (20)4:398-461, noviembre de 2002. Citado en las páginas 529, 531 y 583.

CHAPPELL, D.: *Understanding .NET*. Reading, MA: Addison-Wesley, 2002. Citado en la página 632.

CHERITON, D. y MANN, T.: "Decentralizing a Global Naming Service for Improved Performance and Fault Tolerance." *ACM Trans. Comp. Syst.* (7)2:147-183, mayo de 1989. Citado en la página 203.

CHERITON, D. y SKEEN, D.: "Understanding the Limitations of Causally and Totally Ordered Communication." *Proc. 14th Symp. Operating System Principles*. ACM, 1993, págs. 44-57. Citado en la página 251.

CHERVENAK, A., SCHULER, R., KESSELMAN, C., KORANDA, S. y MOE, B.: "Wide Area Data Replication for Scientific Collaborations." *Proc. Sixth Int'l Workshop on Grid Computing* (Seattle, WA). Nueva York, NY: ACM Press, 2005. Citado en la página 529.

CHERVENAK, A., FOSTER, I., KESSELMAN, C., SALISBURY, C. y TUECKE, S.: "The Data Grid: Towards an Architecture for the Distributed Management and Analysis of Large Scientific Datasets." *J. Netw. Comp. App.* (23)3:187-200, julio de 2000. Citado en la página 380.

CHESWICK, W. y BELLOVIN, S.: *Firewalls and Internet Security*. Reading, MA: Addison-Wesley, 2a. ed., 2000. Citado en la página 418.

CHOW, R. y JOHNSON, T.: *Distributed Operating Systems and Algorithms*. Reading, MA: Addison-Wesley, 1997. Citado en las páginas 363 y 366.

CHUN, B. y SPALINK, T.: "Slice Creation and Managemen.t" Reporte técnico PDN-03-013, Planet Lab Consortium, julio de 2003. Citado en la página 101.

CIANCARINI, P., TOLKSDORF, R., VITALI, F. y KNOCHE, A.: "Coordinating Multiagent Applications on the WWW: A Reference Architecture." *IEEE Trans. Softw. Eng.* (24)5:362-375, mayo de 1998. Citado en la página 610.

CLARK, C., FRASER, K., HAND, S., HANSEN, J. G., JUL, E., LIMPACH, C., PRATT, I. y WARFIELD, A.: "Live Migration of Virtual Machines." *Proc. Second Symp. Networked Systems Design and Impl.* (Boston, MA). Berkeley, CA: USENIX, 2005. Citado en la página 111.

CLARK, D.: "The Design Philosophy of the DARPA Internet Protocols." *Proc. SIGCOMM* (Austin, TX). Nueva York, NY: ACM Press, 1989, págs. 106-114. Citado en la página 91.

CLEMENT, L., HATELY, A., VON RIEGEN, C. y ROGERS, T.: "Universal Description, Discovery and Integration (UDDI)." Reporte técnico, OASIS UDDI, 2004. Citado en la página 222.

COHEN, B.: "Incentives Build Robustness in Bittorrent." *Proc. First Workshop on Economics of Peer-to-Peer Systems* (Berkeley, CA), 2003. Citado en la página 53.

COHEN, D.: "On Holy Wars and a Plea for Peace." *IEEE Computer* (14)10:48-54, octubre de 1981. Citado en la página 131.

COHEN, E. y SHENKER, S.: "Replication Strategies in Unstructured Peer-to-Peer Networks." *Proc. SIGCOMM* (Pittsburgh, PA). Nueva York, NY: ACM Press, 2002, págs. 177-190. Citado en la página 526.

COMER, D.: *Internetworking with TCP/IP, Volume I: Principles, Protocols, and Architecture*. Upper Saddle River, NJ: Prentice Hall, 5a. ed., 2006. Citado en la página 121.

CONTI, M., GREGORI, E. y LAPENNA, W.: "Content Delivery Policies in ReplicatedWeb Services: Client-Side vs. Server-Side." *Cluster Comput.* (8)47-60, enero de 2005. Citado en la página 579.

COPPERSMITH, D.: "The Data Encryption Standard (DES) and its Strength Against Attacks." *IBM J. Research and Development*, (38)3:243-250, mayo de 1994. Citado en la página 394.

COULOURIS, G., DOLLIMORE, J. y KINDBERG, T.: *Distributed Systems, Concepts and Design*. Reading, MA: Addison-Wesley, 4a. ed., 2005. Citado en la página 623.

COX, L. y NOBLE, B.: "Samsara: Honor Among Thieves in Peer-to-Peer Storage." *Proc. 19th Symp. Operating System Principles* (Bolton Landing, NY). Nueva York, NY: ACM Press, 2003, págs. 120-131. Citado en la página 540.

COYLER, A., BLAIR, G. y RASHID, A.: "Managing Complexity In Middleware." *Proc. Second AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software*, 2003. Citado en la página 58.

CRESPO, A. y GARCIA-MOLINA, H.: "Semantic Overlay Networks for P2P Systems." Reporte técnico, Stanford University, Department of Computer Science, 2003. Citado en la página 225.

CRISTIAN, F.: "Probabilistic Clock Synchronization." *Distributed Computing* (3)146-158, 1989. Citado en la página 240.

CRISTIAN, F.: "Understanding Fault-Tolerant Distributed Systems." *Commun. ACM* (34)2:56-78, febrero de 1991. Citado en la página 324.

CRISTIAN, F. y FETZER, C.: "The Timed Asynchronous Distributed System Model." *IEEE Trans. Par. Distr. Syst.* (10)6:642-657, junio de 1999. Citado en la página 629.

CROWLEY, C.: *Operating Systems, A Design-Oriented Approach*. Chicago: Irwin, 1997. Citado en la página 197.

DABEK, F., COX, R., KAASHOEK, F. y MORRIS, R.: "Vivaldi: A Decentralized Network Coordinate System." *Proc. SIGCOMM* (Portland, OR). Nueva York, NY: ACM Press, 2004a. Citado en la página 263.

DABEK, F., KAASHOEK, M. F., KARGER, D., MORRIS, R. y STOICA, I.: "Wide-area Cooperative Storage with CFS." *Proc. 18th Symp. Operating System Principles*. ACM, 2001. Citado en la página 499.

DABEK, F., LI, J., SIT, E., ROBERTSON, J., KAASHOEK, M. F. y MORRIS, R.: "Designing a dht for low latency and high throughput." *Proc. First Symp. Networked Systems Design and Impl.* (San Francisco, CA). Berkeley, CA: USENIX, 2004b, págs. 85-98. Citado en la página 191.

DAIGLE, L., VAN GULIK, D., IANNELLA, R. y FALTSTROM, P.: "Uniform Resource Names (URN) Namespace Definition Mechanisms." RFC 3406, octubre de 2002. Citado en la página 568.

DAVIE, B., CHARNY, A., BENNET, J., BENSON, K., BOUDEC, J. L., COURTNEY, W., S. DAVARI, FIROIU, V. y STILIADIS, D.: "An Expedited Forwarding PHB (Per-Hop Behavior)." RFC 3246, marzo de 2002. Citado en la página 161.

DAY, J. y ZIMMERMAN, H.: "The OSI Reference Model." *Proceedings of the IEEE* (71)12:1334-1340, diciembre de 1983. Citado en la página 117.

DEERING, S., ESTRIN, D., FARINACCI, D., JACOBSON, V., LIU, C.-G. y WEI, L.:"The PIM Architecture for Wide-Area Multicast Routing." *IEEE/ACM Trans. Netw.* (4)2:153-162, abril de 1996. Citado en la página 183.

DEERING, S. y CHERITON, D.: "Multicast Routing in Datagram Internetworks and Extended LANs." *ACM Trans. Comp. Syst.* (8)2:85-110, mayo de 1990. Citado en la página 183.

DEMERS, A., GEHRKE, J., HONG, M., RIEDEWALD, M. y WHITE, W.: "Towards Expressive Publish/Subscribe Systems." *Proc. Tenth Int'l Conf. on Extended Database Technology* (Munich, Alemania), 2006. Citado en la página 607.

DEMERS, A., GREENE, D., HAUSER, C., IRISH, W., LARSON, J., SHENKER, S., STURGIS, H., SWINEHART, D. y TERRY, D.: "Epidemic Algorithms for Replicated Database Maintenance." *Proc. Sixth Symp. on Principles of Distributed Computing* (Vancouver). ACM, 1987, págs. 1-12. Citado en las páginas 170 y 172.

DEUTSCH, P., SCHOULTZ, R., FALTSTROM, P. y WEIDER, C.: "Architecture of the WHOIS++ Service." RFC 1835, agosto de 1995. Citado en la página 63.

D\'FAGO, X., SHIPER, A. y URB{´A}N, P.: "Total Order Broadcast and Multicast Algorithms: Taxonomy and Survey." *ACM Comput. Surv.* (36)4:372-421, diciembre de 2004. Citado en la página 344.

DIAO, Y., HELLERSTEIN, J., PAREKH, S., GRIFFITH, R., KAISER, G. y PHUNG, D.: "A Control Theory Foundation for Self-Managing Computing Systems." *IEEE J. Selected Areas Commun.* (23)12:2213-2222, diciembre de 2005. Citado en la página 60.

DIERKS, T. y ALLEN, C.: "The Transport Layer Security Protocol". RFC 2246, enero de 1996. Citado en la página 584.

DIFFIE, W. y HELLMAN, M.: "New Directions in Cryptography." *IEEE Trans. Information Theory* (IT-22)6:644-654, noviembre de 1976. Citado en la página 429.

DILLEY, J., MAGGS, B., PARIKH, J., PROKOP, H., SITARAMAN, R. y WEIHL, B.: "Globally Distributed Content Delivery." *IEEE Internet Comput.* (6)5:50-58, septiembre de 2002. Citado en la página 577.

DIOT, C., LEVINE, B., LYLES, B., KASSEM, H. y BALENSIEFEN, D.: "Deployment Issues for the IP Multicast Service and Architecture." *IEEE Network* (14)1:78-88, enero de 2000. Citado en la página 166.

DOORN, J. H. y RIVERO, L. C. (eds.): *Database Integrity: Challenges and Solutions*. Hershey, PA: Idea Group, 2002. Citado en la página 384.

DOUCEUR, J. R.: "The Sybil Attack." *Proc. First Int'l Workshop on Peer-to-Peer Systems*, vol. 2429 de *Lect. Notes Comp. Sc*. Berlín: Springer-Verlag, 2002, págs. 251-260. Citado en la página 539.

DUBOIS, M., SCHEURICH, C. y BRIGGS, F.: "Synchronization, Coherence, and Event Ordering in Multiprocessors." *IEEE Computer* (21)2:9-21, febrero de 1988. Citado en la página 283.

DUNAGAN, J., HARVEY, N. J. A., JONES, M. B., KOSTIC, D., THEIMER, M. y WOLMAN, A.: "FUSE: Lightweight Guaranteed Distributed Failure Notification." *Proc. Sixth Symp. on Operating System Design and Implementation* (San Francisco, CA). Berkeley, CA: USENIX, 2004. Citado en la página 336.

DUVVURI, V., SHENOY, P. y TEWARI, R.: "Adaptive Leases: A Strong Consistency Mechanism for the World Wide Web." *IEEE Trans. Know. Data Eng.* (15)5:1266-1276, septiembre de 2003. Citado en la página 304.

EDDON, G. y EDDON, H.: *Inside Distributed COM*. Redmond, WA: Microsoft Press, 1998. Citado en la página 136.

EISLER, M.: "LIPKEY - A Low Infrastructure Public Key Mechanism Using SPKM." RFC 2847, junio de 2000. Citado en la página 534.

EISLER, M., CHIU, A. y LING, L.: "RPCSEC_GSS Protocol Specification." RFC 2203, septiembre de 1997. Citado en la página 534.

ELNOZAHY, E. N. y PLANK, J. S.: "Checkpointing for Peta-Scale Systems: A Look into the Future of Practical Rollback-Recovery." *IEEE Trans. Depend. Secure Comput.* (1)2:97-108, abril de 2004. Citado en la página 368.

ELNOZAHY, E., ALVISI, L., WANG, Y.-M. y JOHNSON, D.: "A Survey of Rollback-Recovery Protocols in Message-Passing Systems." *ACM Comput. Surv.* (34)3:375-408, septiembre de 2002. Citado en las páginas 366 y 372.

ELSON, J., GIROD, L. y ESTRIN, D.: "Fine-Grained Network Time Synchronization using Reference Broadcasts." *Proc. Fifth Symp. on Operating System Design and Implementation* (Boston, MA). Nueva York, NY: ACM Press, 2002, págs. 147-163. Citado en la página 242.

EMMERICH, W.: *Engineering Distributed Objects*. Nueva York: John Wiley, 2000. Citado en la página 631.

EUGSTER, P., FELBER, P., GUERRAOUI, R. y KERMARREC, A.-M.: "The Many Faces of Publish/Subscribe." *ACM Comput. Surv.* (35)2:114-131, junio de 2003. Citado en las páginas 35 y 591.

EUGSTER, P., GUERRAOUI, R., KERMARREC, A.-M. y MASSOULI´E, L.: "Epidemic Information Dissemination in Distributed Systems." *IEEE Computer*, (37)5:60-67, mayo de 2004. Citado en la página 170.

FARMER, W. M., GUTTMAN, J. D. y SWARUP, V.: "Security for Mobile Agents: Issues and Requirements." *Proc. 19th National Information Systems Security Conf.*, 1996, págs. 591-597. Citado en la página 421.

FELBER, P. y NARASIMHAN, P.: "Experiences, Strategies, and Challenges in Building Fault-Tolerant CORBA Systems." *IEEE Computer* (53)5:497-511, mayo de 2004. Citado en la página 479.

FERGUSON, N. y SCHNEIER, B.: *Practical Cryptography*. Nueva York: John Wiley, 2003. Citado en las páginas 391 y 400.

FIELDING, R., GETTYS, J., MOGUL, J., FRYSTYK, H., MASINTER, L., LEACH, P. y BERNERS-LEE, T.: "Hypertext Transfer Protocol –HTTP/1.1." RFC 2616, junio de 1999. Citado en las páginas 122 y 560.

FIELDING, R. T. y TAYLOR, R. N.: "Principled Design of the Modern Web Architecture." *ACM Trans. Internet Techn.* (2)2:115–150, 2002. Citado en la página 633.

FILMAN, R. E., ELRAD, T., CLARKE, S. y AKSIT, M. (eds.): *Aspect-Oriented Software Development*. Reading, MA: Addison-Wesley, 2005. Citado en la página 57.

FISCHER, M., LYNCH, N. y PATTERSON, M.: "Impossibility of Distributed Consensus with one Faulty Processor." *J. ACM* (32)2:374-382, abril de 1985. Citado en la página 334.

FLEURY, M. y REVERBEL, F.: "The JBoss Extensible Server." *Proc. Middleware 2003*, vol. 2672 de *Lect. Notes Comp. Sc.* (Río de Janeiro, Brasil). Berlín: Springer-Verlag, 2003, págs. 344-373. Citado en la página 631.

FLOYD, S., JACOBSON, V., MCCANNE, S., LIU, C. G. y ZHANG, L.: "A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing." *IEEE/ACM Trans. Netw.* (5)6:784-803, diciembre de 1997. Citado en las páginas 345 y 346.

FOSTER, I. y KESSELMAN, C.: *The Grid 2: Blueprint for a New Computing Infrastructure*. San Mateo, CA: Morgan Kaufman, 2a. ed., 2003. Citado en las páginas 380 y 623.

FOSTER, I., KESSELMAN, C., TSUDIK, G. y TUECKE, S.: "A Security Architecture for Computational Grids." *Proc. Fifth Conf. Computer and Communications Security*. ACM, 1998, págs. 83-92. Citado en las páginas 380, 382 y 383.

FOSTER, I., KESSELMAN, C. y TUECKE, S.: "The Anatomy of the Grid, Enabling Scalable Virtual Organizations." *Journal of Supercomputer Applications*, (15)3:200-222, otoño de 2001. Citado en la página 19.

FOSTER, I., KISHIMOTO, H. y SAVVA, A.: "The Open Grid Services Architecture, Version 1.0." GGF Informational Document GFD-I.030, enero de 2005. Citado en la página 20.

FOWLER, R.: *Decentralized Object Finding Using Forwarding Addresses*. Tesis de doctorado, University of Washington, Seattle, 1985. Citado en la página 184.

FRANKLIN, M. J., CAREY, M. J. y LIVNY, M.: "Transactional Client-Server Cache Consistency: Alternatives and Performance." *ACM Trans. Database Syst.* (22)3:315-363, septiembre de 1997. Citado en las páginas 313 y 314.

FREEMAN, E., HUPFER, S. y ARNOLD, K.: *JavaSpaces, Principles, Patterns and Practice*. Reading, MA: Addison-Wesley, 1999. Citado en la página 593.

FREUND, R.: "Web Services Coordination, Version 1.0, febrero de 2005. Citado en la página 553.

FRIEDMAN, R. y KAMA, A.: "Transparent Fault-Tolerant Java Virtual Machine." *Proc. 22nd Symp. on Reliable Distributed Systems* (Florencia, Italia). IEEE Computer Society Press: IEEE Computer Society Press, 2003, págs. 319-328. Citado en las páginas 480 y 481.

FUGGETTA, A., PICCO, G. P. y VIGNA, G.: "Understanding Code Mobility." *IEEE Trans. Softw. Eng.* (24)5:342-361, mayo de 1998. Citado en la página 105.

GAMMA, E., HELM, R., JOHNSON, R. y VLISSIDES, J.: *Design Patterns, Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1994. Citado en las páginas 418 y 446.

GARBACKI, P., EPEMA, D., y VAN STEEN, M.: "A Two-Level Semantic Caching Scheme for Super-Peer Networks." *Proc. Tenth Web Caching Workshop* (Sophia Antipolis, Francia). IEEE, 2005. Citado en la página 51.

GARCIA-MOLINA, H.: "Elections in a Distributed Computing System." *IEEE Trans. Comp.* (31)1:48-59, enero de 1982. Citado en la página 264.

GARMAN, J.: *Kerberos: The Definitive Guide*. Sebastopol, CA: O'Reilly & Associates, 2003. Citado en las páginas 411 y 442.

GELERNTER, D.: "Generative Communication in Linda." *ACM Trans. Prog. Lang. Syst.* (7)1:80-112, 1985. Citado en la página 591.

GELERNTER, D. y CARRIERO, N.: "Coordination Languages and their Significance." *Commun. ACM* (35)2:96-107, febrero de 1992. Citado en la página 590.

GHEMAWAT, S., GOBIOFF, H. y LEUNG, S. T.: "The Google File System." *Proc. 19th Symp. Operating System Principles* (Bolton Landing, NY). Nueva York, NY: ACM Press, 2003, págs. 29-43. Citado en la página 497.

GIFFORD, D.: "Weighted Voting for Replicated Data." *Proc. Seventh Symp. Operating System Principles*. ACM, 1979, págs. 150-162. Citado en la página 311.

GIGASPACES: *GigaSpaces Cache 5.0 Documentation*. Nueva York, NY, 2005. Citado en la página 611.

GIL, T. M. y POLETTO, M.: "MULTOPS: a Data-Structure for Bandwidth Attack Detection." *Proc. Tenth USENIX Security Symp.* (Washington, DC). Berkeley, CA: USENIX, 2001, págs. 23-38. Citado en la página 427.

GLADNEY, H.: "Access Control for Large Collections." *ACM Trans. Inf. Syst.* (15)2:154-194, abril de 1997. Citado en la página 418.

GOLAND, Y., WHITEHEAD, E., FAIZI, A., CARTER, S. y JENSEN, D.: "HTTP Extensions for Distributed Authoring - WEBDAV." RFC 2518, febrero de 1999. Citado en la página 569.

GOLLMANN, D.: *Computer Security*. Nueva York: John Wiley, 2a. ed., 2006. Citado en la página 384.

GONG, L. y SCHEMERS, R.: "Implementing Protection Domains in the Java Development Kit 1.2." *Proc. Symp. Network and Distributed System Security*. Internet Society, 1998, págs. 125-134. Citado en la página 426.

GOPALAKRISHNAN, V., SILAGHI, B., BHATTACHARJEE, B. y KELEHER, P.: "Adaptive Replication in Peer-to-Peer Systems." *Proc. 24th Int'l Conf. on Distributed Computing Systems* (Tokio). Los Alamitos, CA: IEEE Computer Society Press, 2004, págs. 360-369. Citado en la página 527.

GRAY, C. y CHERITON, D.: "Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency." *Proc. 12th Symp. Operating System Principles* (Litchfield Park, AZ). Nueva York, NY: ACM Press, 1989, págs. 202-210. Citado en la página 304.

GRAY, J., HELLAND, P., O'NEIL, P. y SASHNA, D.: "The Dangers of Replication and a Solution." *Proc. SIGMOD Int'l Conf. on Management Of Data*. ACM, 1996, págs. 173-182. Citado en las páginas 276 y 628.

GRAY, J. y REUTER, A.: *Transaction Processing: Concepts and Techniques*. San Mateo, CA: Morgan Kaufman, 1993. Citado en la página 21.

GRAY, J.: "Notes on Database Operating Systems." En Bayer, R., Graham, R. y Seegmuller, G. (eds.), *Operating Systems: An Advanced Course*, vol. 60 de *Lect. Notes Comp. Sc.*, págs. 393-481. Berlín: Springer-Verlag, 1978. Citado en la página 355.

GRIMM, R., DAVIS, J., LEMAR, E., MACBETH, A., SWANSON, S., ANDERSON, T., BERSHAD, B., BORRIELLO, G., GRIBBLE, S. y WETHERALL, D.: "System Support for Pervasive Applications." *ACM Trans. Comp. Syst.* (22)4:421-486, noviembre de 2004. Citado en la página 25.

GROPP, W., HUSS-LEDERMAN, S., LUMSDAINE, A., LUSK, E., NITZBERG, B., SAPHIR, W. y SNIR, M.: *MPI: The Complete Reference – The MPI-2 Extensions*. Cambridge, MA: MIT Press, 1998a. Citado en la página 145.

GROPP, W., LUSK, E. y SKJELLUM, A.: *Using MPI, Portable Parallel Programming with the Message-Passing Interface*. Cambridge, MA: MIT Press, 2a. ed., 1998b. Citado en la página 145.

GROSSKURTH, A. y GODFREY, M. W.: "A Reference Architecture for Web Browsers." *Proc. 21st Int'l Conf. Softw. Mainten.* (Budapest, Hungría). Los Alamitos, CA: IEEE Computer Society Press, 2005, págs. 661-664. Citado en la página 554.

GUDGIN, M., HADLEY, M., MENDELSOHN, N., MOREAU, J. J. y NIELSEN, H. F.: "SOAP Version 1.2." W3C Recommendation, junio de 2003. Citado en las páginas 565 y 567.

GUERRAOUI, R. y RODRIGUES, L.: *Introduction to Reliable Distributed Programming*. Berlín: Springer-Verlag, 2006. Citado en las páginas 232 y 627.

GUERRAOUI, R. y SCHIPER, A.: "Software-Based Replication for Fault Tolerance." *IEEE Computer* (30)4:68-74, abril de 1997. Citado en las páginas 328 y 629.

GUICHARD, J., FAUCHEUR, F. L. y VASSEUR, J.-P.: *Definitive MPLS Network Designs*. Indianápolis, IN: Cisco Press, 2005. Citado en la página 575.

GULBRANDSEN, A., VIXIE, P. y ESIBOV, L.: "A dns rr for specifying the location of services (dns srv)." RFC 2782, febrero de 2000. Citado en la página 211.

GUPTA, A., SAHIN, O. D., AGRAWAL, D. y ABBADI, A. E.: "Meghdoot: Content-Based Publish/Subscribe over P2P Networks." *Proc. Middleware 2004*, vol. 3231 de *Lect. Notes Comp. Sc.* (Toronto, Canadá). Berlín: Springer-Verlag, 2004, págs. 254-273. Citado en la página 599.

GUSELLA, R. y ZATTI, S.: "The Accuracy of the Clock Synchronization Achieved by TEMPO in Berkeley UNIX 4.3BSD". *IEEE Trans. Softw. Eng.* (15)7:847-853, julio de 1989. Citado en la página 241.

HADZILACOS, V. y TOUEG, S.: "Fault-Tolerant Broadcasts and Related Problems." En Mullender, S. (ed.), *Distributed Systems*, págs. 97-145. Wokingham: Addison-Wesley, 2a. ed., 1993. Citado en las páginas 324 y 352.

HALSALL, F.: *Multimedia Communications: Applications, Networks, Protocols and Standards*. Reading, MA: Addison-Wesley, 2001. Citado en las páginas 157 y 160.

HANDURUKANDE, S., KERMARREC, A.-M., FESSANT, F. L. y MASSOULI´E, L.:“Exploiting Semantic Clustering in the eDonkey P2P network.” *Proc. 11th SIGOPS European Workshop* (Leuven, Bélgica). Nueva York, NY: ACM Press, 2004. Citado en la página 226.

HELDER, D. A. y JAMIN, S.: “End-Host Multicast Communication Using Switch-Trees Protocols.” *Proc. Second Int'l Symp. Cluster Comput. & Grid* (Berlín, Alemania). Los Alamitos, CA: IEEE Computer Society Press, 2002, págs. 419-424. Citado en la página 169.

HELLERSTEIN, J. L., DIAO, Y., PAREKH, S. y TILBURY, D. M.: *Feedback Control of Computing Systems*. Nueva York: John Wiley, 2004. Citado en las páginas 60 y 624.

HENNING, M.: “A New Approach to Object-Oriented Middleware.” *IEEE Internet Comput.*, (8)1:66-75, enero de 2004. Citado en la página 454.

HENNING, M.: “The Rise and Fall of CORBA.” *ACM Queue* (4)5, 2006. Citado en la página 631.

HENNING, M. y SPRUIELL, M.: *Distributed Programming with Ice*. ZeroC Inc., Brisbane, Australia, mayo de 2005. Citado en las páginas 455 y 470.

HENNING, M. y VINOSKI, S.: *Advanced CORBA Programming with C++*. Reading, MA: Addison-Wesley, 1999. Citado en la página 631.

HOCHSTETLER, S. y BERINGER, B.: “Linux Clustering with CSM and GPFS.” Reporte técnico SG24-6601-02, International Technical Support Organization, IBM, Austin, TX, enero de 2004. Citado en la página 98.

HOHPE, G. y WOOLF, B.: *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Reading, MA: Addison-Wesley, 2004. Citado en las páginas 152 y 626.

HOROWITZ, M. y LUNT, S.: “FTP Security Extensions.” RFC 2228, octubre de 1997. Citado en la página 122.

HOWES, T.: “The String Representation of LDAP Search Filters.” RFC 2254, diciembre de 1997. Citado en la página 221.

HUA CHU, Y., RAO, S. G., SESHAN, S. y ZHANG, H.: “A Case for End System Multicast.” *IEEE J. Selected Areas Commun.* (20)8:1456-1471, octubre de 2002. Citado en la página 168.

HUFFAKER, B., FOMENKOV, M., PLUMMER, D. J., MOORE, D. y CLAFFY, K.: “Distance Metrics in the Internet.” *Proc. Int'l Telecommun. Symp.* (Natal RN, Brasil). Los Alamitos, CA: IEEE Computer Society Press, 2002. Citado en la página 575.

HUNT, G., NAHUM, E. y TRACEY, J.: “Enabling Content-Based Load Distribution for Scalable Services.” Reporte técnico, IBM T.J. Watson Research Center, mayo de 1997. Citado en la página 94.

HUTTO, P. y AHAMAD, M.: “Slow Memory: Weakening Consistency to Enhance Concurrency in Distributed Shared Memories.” *Proc. Tenth Int'l Conf. on Distributed Computing Systems*. IEEE, 1990, págs. 302-311. Citado en la página 284.

IBM: *WebSphere MQ Application Programming Guide*, mayo de 2005a. Citado en la página 152.

IBM: *WebSphere MQ Intercommunication*, mayo de 2005b. Citado en la página 152.

IBM: *WebSphere MQ Publish/Subscribe User's Guide*, mayo de 2005c. Citado en la página 593.

IBM: *WebSphere MQ System Administration*, mayo de 2005d. Citado en la página 152.

ISO: "Open Distributed Processing Reference Model." International Standard ISO/IEC IS 10746, 1995. Citado en la página 5.

JAEGER, T., PRAKASH, A., LIEDTKE, J. e ISLAM, N.: "Flexible Control of Downloaded Executable Content." *ACM Trans. Inf. Syst. Sec.* (2)2:177-228, mayo de 1999. Citado en la página 426.

JALOTE, P.: *Fault Tolerance in Distributed Systems*. Englewood Cliffs, NJ: Prentice Hall, 1994. Citado en las páginas 312, 322 y 630.

JANIC, M.: *Multicast in Network and Application Layer*. Tesis de doctorado, Delft University of Technology, Países Bajos, octubre de 2005. Citado en la página 166.

JANIGA, M. J., DIBNER, G. y GOVERNALI, F. J.: "Internet Infrastructure: Content Delivery." Goldman Sachs Global Equity Research, abril de 2001. Citado en la página 575.

JELASITY, M., GUERRAOUI, R., KERMARREC, A. M. y VAN STEEN, M.: "The Peer Sampling Service: Experimental Evaluation of Unstructured Gossip-Based Implementations." *Proc. Middleware 2004*, vol. 3231 de *Lect. Notes Comp. Sc.* (Toronto, Canadá). Berlín: Springer-Verlag, 2004, págs. 79-98. Citado en la página 47.

JELASITY, M., VOULGARIS, S., GUERRAOUI, R., KERMARREC, A. M. y VAN STEEN, M.: "Gossip-based Peer Sampling." Reporte técnico, Vrije Universiteit, Department of Computer Science, septiembre de 2005a. Citado en las páginas 47, 49, 171 y 226.

JELASITY, M. y BABAOGLU, O.: "T-Man: Gossip-based Overlay Topology Management." *Proc. Third Int'l Workshop Eng. Self-Organising App.* (Utrecht, Países Bajos), 2005. Citado en las páginas 49 y 50.

JELASITY, M., MONTRESOR, A. y BABAOGLU, O.: "Gossip-based Aggregation in Large Dynamic Networks." *ACM Trans. Comp. Syst.* (23)3:219-252, agosto de 2005b. Citado en la página 173.

JIN, J. y NAHRSTEDT, K.: "QoS Specification Languages for Distributed Multimedia Applications: A Survey and Taxonomy." *IEEE Multimedia* (11)3:74-87, julio de 2004. Citado en la página 160.

JING, J., HELAL, A. y ELMAGARMID, A.: "Client-Server Computing in Mobile Environments." *ACM Comput. Surv.* (31)2:117-157, junio de 1999. Citado en la página 41.

JOHNSON, B.: "An Introduction to the Design and Analysis of Fault-Tolerant Systems." En Pradhan, D.K. (ed.), *Fault-Tolerant Computer System Design*, págs. 1-87. Upper Saddle River, NJ: Prentice Hall, 1995. Citado en la página 326.

JOHNSON, D., PERKINS, C. y ARKKO, J.: "Mobility Support for IPv6." RFC 3775, junio de 2004. Citado en la página 186.

JOSEPH, J., ERNEST, M. y FELLENSTEIN, C.: "Evolution of grid computing architecture and grid adoption models." *IBM Syst. J.* (43)4:624-645, abril de 2004. Citado en la página 20.

JUL, E., LEVY, H., HUTCHINSON, N. y BLACK, A.: "Fine-Grained Mobility in the Emerald System." *ACM Trans. Comp. Syst.* (6)1:109-133, febrero de 1988. Citado en la página 186.

JUNG, J., SIT, E., BALAKRISHNAN, H. y MORRIS, R.: "DNS Performance and the Effectiveness of Caching." *IEEE/ACM Trans. Netw.* (10)5:589 - 603, octubre de 2002. Citado en la página 216.

KAHN, D.: *The Codebreakers*. Nueva York: Macmillan, 1967. Citado en la página 391.

KAMINSKY, M., SAVVIDES, G., MAZIhRES, D. y KAASHOEK, M. F.: "Decentralized User Authentication in a Global File System." *Proc. 19th Symp. Operating System Principles* (Bolton Landing, NY). Nueva York, NY: ACM Press, 2003, págs. 60-73. Citado en las páginas 535 y 538.

KANTARCIOGLU, M. y CLIFTON, C.: "Security Issues in Querying Encrypted Data." *Proc. 19th Conf. Data & Appl. Security*, vol. 3654 de *Lect. Notes Comp. Sc.* (Storrs, CT). Berlín: Springer-Verlag, 2005, págs. 325-337. Citado en la página 618.

KARNIK, N. y TRIPATHI, A.: "Security in the Ajanta Mobile Agent System." *Software – Practice & Experience* (31)4:301-329, abril de 2001. Citado en la página 421.

KASERA, S., KUROSE, J. y TOWSLEY, D.: "Scalable Reliable Multicast Using Multiple Multicast Groups." *Proc. Int'l Conf. Measurements and Modeling of Computer Systems*. ACM, 1997. págs, 64-74. Citado en la página 346.

KATZ, E., BUTLER, M. y MCGRATH, R.: "A Scalable HTTP Server: The NCSA Prototype." *Comp. Netw. & ISDN Syst.* (27)2:155-164, septiembre de 1994. Citado en la página 76.

KAUFMAN, C., PERLMAN, R. y SPECINER, M.: *Network Security: Private Communication in a Public World*. Englewood Cliffs, NJ: Prentice Hall, 2a. ed., 2003. Citado en las páginas 400 y 630.

KENT, S.: "Internet Privacy Enhanced Mail". *Commun. ACM* (36)8:48-60, agosto de 1993. Citado en la página 431.

KEPHART, J. O. y CHESS, D. M.: "The Vision of Autonomic Computing." *IEEE Computer* (36)1:41-50, enero de 2003. Citado en la página 59.

KHOSHAFIAN, S. y BUCKIEWICZ, M.: *Introduction to Groupware, Workflow, and Workgroup Computing*. Nueva York: John Wiley, 1995. Citado en la página 151.

KHURANA, H. y KOLEVA, R.: "Scalable Security and Accounting Services for Content-Based Publish Subscribe Systems." *Int'l J. E-Business Res.* (2), 2006. Citado en las páginas 618, 619 y 620.

KIM, S., PAN, K., SINDERSON, E. y WHITEHEAD, J.: "Architecture and Data Model of a WebDAV-based Collaborative System." *Proc. Collaborative Techn. Symp.\fR* (San Diego, CA), 2004, págs. 48-55. Citado en la página 570.

KISTLER, J. y SATYANARYANAN, M.: "Disconnected Operation in the Coda File System." *ACM Trans. Comp. Syst.* (10)1:3-25, febrero de 1992. Citado en las páginas 503 y 518.

KLEIMAN, S.: "Vnodes: an Architecture for Multiple File System Types in UNIX." *Proc. Summer Techn. Conf.* USENIX, 1986, págs. 238-247. Citado en la página 493.

KOHL, J., NEUMAN, B. y T'SO, T.: "The Evolution of the Kerberos Authentication System." En F. Brazier y D. Johansen (eds.), *Distributed Open Systems*, págs. 78-94. Los Alamitos, CA: IEEE Computer Society Press, 1994. Citado en la página 411.

KON, F., COSTA, F., CAMPBELL, R. y BLAIR, G.: "The Case for Reflective Middleware." *Commun. ACM* (45)6:33-38, junio de 2002. Citado en la página 57.

KOPETZ, H. y VERISSIMO, P.: "Real Time and Dependability Concepts." En S. Mullender (ed.), *Distributed Systems*, págs. 411-446. Wokingham: Addison-Wesley, 2a. ed., 1993. Citado en la página 322.

KOSTOULAS, M. G., MATSA, M., MENDELSOHN, N., PERKINS, E., HEIFETS, A. y MERCALDI, M.: "XML Screamer: An Integrated Approach to High Performance XML Parsing, Validation and Deserialization." *Proc. 15th Int'l WWW Conf.* (Edinburgo, Escocia). Nueva York, NY: ACM Press, 2006. Citado en la página 567.

KUMAR, P. y SATYANARAYANAN, M.: "Flexible and Safe Resolution of File Conflicts." *Proc. Winter Techn. Conf.* USENIX, 1995, págs. 95-106. Citado en la página 526.

LAI, A. y NIEH, J.: "Limits of Wide-Area Thin-Client Computing." *Proc. Int'l Conf. Measurements and Modeling of Computer Systems* (Marina Del Rey, CA). Nueva York, NY: ACM Press, 2002, págs. 228-239. Citado en la página 84.

LAMACCHIA, B. y ODLYZKO, A.: "Computation of Discrete Logarithms in Prime Fields." *Designs, Codes, and Cryptography* (1)1:47-62, mayo de 1991. Citado en la página 534.

LAMPORT, L.: "Time, Clocks, and the Ordering of Events in a Distributed System." *Commun. ACM* (21)7:558-565, julio de 1978. Citado en la página 244.

LAMPORT, L.: "How to Make a Multiprocessor Computer that Correctly Executes Multiprocessor Programs." *IEEE Trans. Comp.* (C-29)9:690-691, septiembre de 1979. Citado en la página 282.

LAMPORT, L., SHOSTAK, R. y PAESE, M.: "Byzantine Generals Problem." *ACM Trans. Prog. Lang. Syst.* (4)3:382-401, julio de 1982. Citado en las páginas 326, 332 y 334.

LAMPSON, B., ABADI, M., BURROWS, M. y WOBBER, E.: "Authentication in Distributed Systems: Theory and Practice." *ACM Trans. Comp. Syst.* (10)4:265-310, noviembre de 1992. Citado en la página 397.

LAPRIE, J.-C.: "Dependability – Its Attributes, Impairments and Means." En B. Rendell, J. C. Laprie, H. Kopetz, y B. Littlewood, (eds.), *Predictably Dependable Computing Systems*, págs. 3-24. Berlín: Springer-Verlag, 1995. Citado en la página 378.

LAURIE, B. y LAURIE, P.: *Apache: The Definitive Guide*. Sebastopol, CA: O'Reilly & Associates, 3a. ed., 2002. Citado en la página 558.

LEFF, A. y RAYFIELD, J. T.: "Alternative Edge-server Architectures for Enterprise JavaBeans Applications." *Proc. Middleware 2004*, vol. 3231 de *Lect. Notes Comp. Sc.* (Toronto, Canadá). Berlín: Springer-Verlag, 2004, págs. 195-211. Citado en la página 52.

LEIGHTON, F. y LEWIN, D.: "Global Hosting System." Patente estadounidense número 6, 108,703, agosto de 2000. Citado en la página 577.

LEVIEN, R. (ed.): *Signposts in Cyberspace: The Domain Name System and Internet Navigation*. Washington, DC: National Academic Research Council, 2005. Citado en la página 210.

LEVINE, B. y GARCÍA-LUNA-ACEVES, J.: "A Comparison of Reliable Multicast Protocols." *ACM Multimedia Systems Journal*, (6)5:334-348, 1998. Citado en la página 345.

LEWIS, B. y BERG, D. J.: *Multithreaded Programming with Pthreads*. Englewood Cliffs, NJ: Prentice Hall, 2a. ed., 1998. Citado en las páginas 70 y 625.

LI, G. y JACOBSEN, H. A.: "Composite Subscriptions in Content-Based Publish/Subscribe Systems." *Proc. Middleware 2005*, vol. 3790 de *Lect. Notes Comp. Sc.* (Grenoble, Francia). Berlín: Springer-Verlag, 2005, págs. 249-269. Citado en la página 603.

LI, J., LU, C. y SHI, W.: "An Efficient Scheme for Preserving Confidentiality in Content-Based Publish-Subscribe Systems." Reporte técnico GIT-CC-04-01, Georgia Institute of Technology, College of Computing, 2004a. Citado en la página 619.

LI, N., MITCHELL, J. C. y TONG, D.: "Securing Java RMI-based Distributed Applications." *Proc. 20th Ann. Computer Security Application Conf.* (Tucson, AZ). ACSA, 2004b. Citado en la página 486.

LILJA, D.: "Cache Coherence in Large-Scale Shared-Memory Multiprocessors: Issues and Comparisons." *ACM Comput. Surv.* (25)3:303-338, septiembre de 1993. Citado en la página 313.

LIN, M. J. y MARZULLO, K.: "Directional Gossip: Gossip in a Wide-Area Network." En *Proc. Third European Dependable Computing Conf.*, vol. 1667 de *Lect. Notes Comp. Sc.*, págs. 364-379. Berlín: Springer-Verlag, septiembre de 1999. Citado en la página 172.

LIN, S.-D., LIAN, Q., CHEN, M. y ZHANG, Z.: "A Practical Distributed Mutual Exclusion Protocol in Dynamic Peer-to-Peer Systems." *Proc. Third Int'l Workshop on Peer-to-Peer Systems*, vol. 3279 de *Lect. Notes Comp. Sc.* (La Jolla, CA). Berlín: Springer-Verlag, 2004, págs. 11-21. Citado en las páginas 254 y 255.

LING, B. C., KICIMAN, E. y FOX, A.: "Session State: Beyond Soft State." *Proc. First Symp. Networked Systems Design and Impl.* (San Francisco, CA). Berkeley, CA: USENIX, 2004, págs. 295-308. Citado en la página 91.

LINN, J.: "Generic Security Service Application Program Interface, version 2." RFC 2078, enero de 1997. Citado en la página 534.

LIU, C.-G., ESTRIN, D., SHENKER, S. y ZHANG, L.: "Local Error Recovery in SRM: Comparison of Two Approaches." *IEEE/ACM Trans. Netw.*, (6)6:686-699, diciembre de 1998. Citado en la página 346.

LIU, H. y JACOBSEN, H. A.: "Modeling Uncertainties in Publish/Subscribe Systems." *Proc. 20th Int'l Conf. Data Engineering* (Boston, MA). Los Alamitos, CA: IEEE Computer Society Press, 2004. págs, 510-522. Citado en la página 607.

LO, V., ZHOU, D., LIU, Y., GauthierDickey, C. y LI, J.: "Scalable Supernode Selection in Peer-to-Peer Overlay Networks." *Proc. Second Hot Topics in Peer-to-Peer Systems*, (La Jolla, CA), 2005. Citado en la página 269.

LOSHIN, P. (ed.): *Big Book of Lightweight Directory Access Protocol (LDAP) RFCs*. San Mateo, CA: Morgan Kaufman, 2000. Citado en la página 627.

LUA, E. K., CROWCROFT, J., PIAS, M., SHARMA, R. y LIM, S.: "A Survey and Comparison of Peer-to-Peer Overlay Network Schemes." *IEEE Communications Surveys & Tutorials* (7)2:22-73, abril de 2005. Citado en las páginas 15, 44 y 625.

LUI, J., MISRA, V. y RUBENSTEIN, D.: "On the Robustness of Soft State Protocols." *Proc. 12th Int'l Conf. on Network Protocols*, (Berlín, Alemania). Los Alamitos, CA: IEEE Computer Society Press, 2004, págs. 50-60. Citado en la página 91.

LUOTONEN, A. y ALTIS, K.: "World-Wide Web Proxies." *Comp. Netw. & ISDN Syst.* (27)2:1845-1855, 1994. Citado en la página 555.

LYNCH, N.: *Distributed Algorithms*. San Mateo, CA: Morgan Kaufman, 1996. Citado en las págs. 232, 263 y 628.

MAASSEN, J., KIELMANN, T. y BAL, H. E.: "Parallel Application Experience with Replicated Method Invocation." *Conc. & Comput.: Prac. Exp.* (13)8-9:681-712, 2001. Citado en la página 475.

MACGREGOR, R., DURBIN, D., OWLETT, J. y YEOMANS, A.: *Java Network Security*. Upper Saddle River, NJ: Prentice Hall, 1998. Citado en la página 422.

MADDEN, S. R., FRANKLIN, M. J., HELLERSTEIN, J. M. y HONG, W.: "TinyDB: An Acquisitional Query Processing System for Sensor Networks." *ACM Trans. Database Syst.* (30)1:122-173, 2005. Citado en la página 30.

MAKPANGOU, M., GOURHANT, Y., LE NARZUL, J. P. y SHAPIRO, M.: "Fragmented Objects for Distributed Abstractions." En T. Casavant y M. Singhal (eds.), *Readings in Distributed Computing Systems*, págs. 170-186. Los Alamitos, CA: IEEE Computer Society Press, 1994. Citado en la página 449.

MALKHI, D. y REITER, M.: "Secure Execution of Java Applets using a Remote Playground." *IEEE Trans. Softw. Eng.*, (26)12:1197-1209, diciembre de 2000. Citado en la página 424.

MAMEI, M. y ZAMBONELLI, F.: "Programming Pervasive and Mobile Computing Applications with the TOTA Middleware." *Proc. Second Int'l Conf. Pervasive Computing and Communications* (*PerCom*) (Orlando, FL). Los Alamitos, CA: IEEE Computer Society Press, 2004, págs. 263-273. Citado en la página 601.

MANOLA, F. y MILLER, E.: "RDF Primer." W3C Recommendation, febrero de 2004. Citado en la página 218.

MARCUS, E. y STERN, H.: *Blueprints for High Availability*. Nueva York: John Wiley, 2a. ed., 2003. Citado en la página 629.

MASCOLO, C., CAPRA, L. y EMMERICH, W.: "Principles of Mobile Computing Middleware." En Qusay H. Mahmoud (ed.), *Middleware for Communications*, cap. 12. Nueva York: John Wiley, 2004. Citado en la página 25.

MASINTER, L.: "The Data URL Scheme." RFC 2397, agosto de 1998. Citado en la página 568.

MAZIERES, D., KAMINSKY, M., KAASHOEK, M. y WITCHEL, E.: "Separating Key Management from File System Security." *Proc. 17th Symp. Operating System Principles*. ACM, 1999, págs. 124-139. Citado en las páginas 484 y 536.

MAZOUNI, K., GARBINATO, B. y GUERRAOUI, R.: "Building Reliable Client-Server Software Using Actively Replicated Objects." En I. Graham, B. Magnusson, B. Meyer, y J.-M Nerson (eds.), *Technology of Object Oriented Languages and Systems*, págs. 37-53. Englewood Cliffs, NJ: Prentice Hall, 1995. Citado en la página 475.

MCKINLEY, P., SADJADI, S., KASTEN, E. y CHENG, B.: "Composing Adaptive Software." *IEEE Computer* (37)7:56-64, enero de 2004. Citado en la página 57.

MEHTA, N., MEDVIDOVIC, N. y PHADKE, S.: "Towards A Taxonomy Of Software Connectors." *Proc. 22nd Int'l Conf. on Software Engineering*, (Limerick, Irlanda). Nueva York, NY: ACM Press, 2000, págs. 178-187. Citado en la página 34.

MENEZES, A. J., VAN OORSCHOT, P. C. y VANSTONE, S. A.: *Handbook of Applied Cryptography*. Boca Raton: CRC Press, 3a. ed., 1996. Citado en las páginas 391, 430, 431 y 630.

MERIDETH, M. G., IYENGAR, A., MIKALSEN, T., TAI, S., ROUVELLOU, I. y NARASIMHAN, P.: "Thema: Byzantine-Fault-Tolerant Middleware for Web-Service Applications." *Proc. 24th Symp. on Reliable Distributed Systems* (Orlando, FL). Los Alamitos, CA: IEEE Computer Society Press, 2005, págs. 131-142. Citado en la página 583.

MEYER, B.: *Object-Oriented Software Construction*. Englewood Cliffs, NJ: Prentice Hall, 2a. ed., 1997. Citado en la página 445.

MILLER, B. N., KONSTAN, J. A. y RIEDL, J.: "PocketLens: Toward a Personal Recommender System." *ACM Trans. Inf. Syst.* (22)3:437-476, julio de 2004. Citado en la página 27.

MILLS, D. L.: *Computer Network Time Synchronization: The Network Time Protocol*. Boca Raton, FL: CRC Press, 2006. Citado en la página 241.

MILLS, D. L.: "Network Time Protocol (version 3): Specification, Implementation, and Analysis." RFC 1305, julio de 1992. Citado en la página 241.

MILOJICIC, D., DOUGLIS, F., PAINDAVEINE, Y., WHEELER, R. y ZHOU, S.: "Process Migration." *ACM Comput. Surv.* (32)3:241-299, septiembre de 2000. Citado en la página 103.

MIN, S. L. y BAER, J.-L.: "Design and Analysis of a Scalable Cache Coherence Scheme Based on Clocks and Timestamps." *IEEE Trans. Par. Distr. Syst.* (3)1:25-44, enero de 1992. Citado en la página 313.

MIRKOVIC, J., DIETRICH, S. y AND PETER REIHER, D. D.: *Internet Denial of Service: Attack and Defense Mechanisms*. Englewood Cliffs, NJ: Prentice Hall, 2005. Citado en la página 428.

MIRKOVIC, J. y REIHER, P.: "A Taxonomy of DDoS Attack and DDoS Defense Mechanisms." *ACM Comp. Commun. Rev.* (34)2:39-53, abril de 2004. Citado en la página 428.

MOCKAPETRIS, P.: "Domain Names - Concepts and Facilities." RFC 1034, noviembre de 1987. Citado en las páginas 203 y 210.

MONSON-HAEFEL, R., BURKE, B. y LABOUREY, S.: *Enterprise Java Beans*. Sebastopol, CA: O'Reilly & Associates, 4a. ed., 2004. Citado en la página 447.

MOSER, L., MELLIAR-SMITH, P., AGARWAL, D., BUDHIA, R. y LINGLEYPAPADOPOULOS, C.: "Totem: A Fault-Tolerant Multicast Group Communication System." *Commun. ACM* (39)4:54-63, abril de 1996. Citado en la página 478.

MOSER, L., MELLIOR-SMITH, P. y NARASIMHAN, P.: "Consistent Object Replication in the Eternal System." *Theory and Practice of Object Systems* (4)2:81-92, 1998. Citado en la página 478.

MULLENDER, S. y TANENBAUM, A.: "Immediate Files." *Software - Practice & Experience* (14)3:365-368, 1984. Citado en la página 568.

MUNTZ, D. y HONEYMAN, P.: "Multi-level Caching in Distributed File Systems." *Proc. Winter Techn. Conf.* USENIX, 1992, págs. 305-313. Citado en la página 301.

MURPHY, A., PICCO, G. y ROMAN, G.-C.: "Lime: A Middleware for Physical and Logical Mobility." *Proc. 21st Int'l Conf. on Distr. Computing Systems*, (Phoenix, AZ). Los Alamitos, CA: IEEE Computer Society Press, 2001, págs. 524-533. Citado en la página 600.

MUTHITACHAROEN, A., MORRIS, R., GIL, T. y CHEN, B.: "Ivy: A Read/Write Peer-to-Peer File System." *Proc. Fifth Symp. on Operating System Design and Implementation* (Boston, MA). Nueva York, NY: ACM Press, 2002, págs. 31-44. Citado en la página 499.

NAPPER, J., ALVISI, L. y VIN, H. M.: "A Fault-Tolerant Java Virtual Machine." *Proc. Int'l Conf. Dependable Systems and Networks* (San Francisco, CA). Los Alamitos, CA: IEEE Computer Society Press, 2003, págs. 425-434. Citado en la página 480.

NARASIMHAN, P., MOSER, L. y MELLIAR-SMITH, P.: "The Eternal System." En J. Urban y P. Dasgupta (eds.), *Encyclopedia of Distributed Computing*. Dordrecht, Países Bajos,: Kluwer Academic Publishers, 2000. Citado en la página 478.

NAYATE, A., DAHLIN, M. e IYENGAR, A.: "Transparent Information Dissemination." *Proc. Middleware 2004*, vol. 3231 de *Lect. Notes Comp. Sc.* (Toronto, Canadá). Berlín: Springer-Verlag, 2004, págs. 212-231. Citado en la página 52.

NEEDHAM, R. y SCHROEDER, M.: "Using Encryption for Authentication in Large Networks of Computers." *Commun. ACM* (21)12:993-999, diciembre de 1978. Citado en la página 402.

NEEDHAM, R.: "Names." En S. Mullender (ed.), *Distributed Systems*, págs. 315-327. Wokingham: Addison-Wesley, 2a. ed., 1993. Citado en la página 627.

NELSON, B.: *Remote Procedure Call*. Tesis de doctorado, Carnegie-Mellon University, 1981. Citado en la página 342.

NEUMAN, B.: "Scale in Distributed Systems." En T. y M. Singhal (eds.), *Readings in Distributed Computing Systems*, págs. 463-489. Los Alamitos, CA: IEEE Computer Society Press, 1994. Citado en las páginas 9, 12 y 624.

NEUMAN, B.: "Proxy-Based Authorization and Accounting for Distributed Systems." *Proc. 13th Int'l Conf. on Distributed Computing Systems*. IEEE, 1993, págs. 283-291. Citado en la página 437.

NEUMAN, C., YU, T., HARTMAN, S. y RAEBURN, K.: "The Kerberos Network Authentication Service." RFC 4120, julio de 2005. Citado en la página 411.

NEUMANN, P.: "Architectures and Formal Representations for Secure Systems." Reporte técnico, Computer Science Laboratory, SRI International, Menlo Park, CA, octubre de 1995. Citado en la página 388.

NG, E. y ZHANG, H.: "Predicting Internet Network Distance with Coordinates-Based Approaches." *Proc. 21st INFOCOM Conf.* (Nueva York, NY). Los Alamitos, CA: IEEE Computer Society Press, 2002. Citado en la página 262.

NIEMELA, E. y LATVAKOSKI, J.: "Survey of Requirements and Solutions for Ubiquitous Software." *Proc. Third Int'l Conf. Mobile & Ubiq. Multimedia* (College Park, MY), 2004, págs. 71-78. Citado en la página 25.

NOBLE, B., FLEIS, B. y KIM, M.: "A Case for Fluid Replication." *Proc. NetStore'99*, 1999. Citado en la página 301.

OBRACZKA, K.: "Multicast Transport Protocols: A Survey and Taxonomy." *IEEE Commun. Mag.* (36)1:94-102, enero de 1998. Citado en la página 166.

OMG: "The Common Object Request Broker: Core Specification, revision 3.0.3." OMG Document formal/04-03-12, Object Management Group, Framingham, MA, marzo de 2004a. Citado en las páginas 54, 454, 465 y 477.

OMG: "UML 2.0 Superstructure Specification." OMG Document ptc/04-10-02, Object Management Group, Framingham, MA, octubre de 2004b. Citado en la página 34.

OPPENHEIMER, D., ALBRECHT, J., PATTERSON, D. y VAHDAT, A.: "Design and Implementation Tradeoffs for Wide-Area Resource Discovery." *Proc. 14th Int'l Symp. On High Performance Distributed Computing* (Research Triangle Park, NC). Los Alamitos, CA: IEEE Computer Society Press, 2005. Citado en la página 224.

ORAM, A. (ed.): *Peer-to-Peer: Harnessing the Power of Disruptive Technologies*. Sebastopol, CA: O'Reilly & Associates, 2001. Citado en las páginas 15 y 625.

OZSU, T. y VALDURIEZ, P.: *Principles of Distributed Database Systems*. Upper Saddle River, NJ: Prentice Hall, 2a. ed., 1999. Citado en las páginas 43 y 298.

PAI, V., ARON, M., BANGA, G., SVENDSEN, M., DRUSCHEL, P., ZWAENEPOEL, W. y NAHUM, E.: "Locality-Aware Request Distribution in Cluster-Based Network Servers." *Proc. Eighth Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, (San Jose CA). Nueva York, NY: ACM Press, 1998, págs. 205-216. Citado en la página 94.

PANZIERI, F. y SHRIVASTAVA, S.: "Rajdoot: A Remote Procedure Call Mechanism with Orphan Detection and Killing." *IEEE Trans. Softw. Eng.* (14)1:30-37, enero de 1988. Citado en la página 342.

PARTRIDGE, C., MENDEZ, T. y MILLIKEN, W.: "Host Anycasting Service." RFC 1546, noviembre de 1993. Citado en la página 228.

PATE, S.: *UNIX Filesystems: Evolution, Design, and Implementation*. Nueva York: John Wiley, 2003. Citado en la página 631.

PEASE, M., SHOSTAK, R. y LAMPORT, L.: "Reaching Agreement in the Presence of Faults." *J. ACM* (27)2:228-234, abril de 1980. Citado en la página 326.

PERKINS, C., HODSON, O. y HARDMAN, V.: "A Survey of Packet Loss Recovery Techniques for Streaming Audio." *IEEE Network* (12)5:40-48, septiembre de 1998. Citado en la página 162.

PETERSON, L. y DAVIE, B.: *Computer Networks, A Systems Approach*. San Mateo, CA: Morgan Kaufman, 3a. ed., 2003. Citado en la página 626.

PETERSON, L., BAVIER, A., FIUCZYNSKI, M., MUIR, S. y ROSCOE, T.: "Towards a Comprehensive PlanetLab Architecture." Reporte técnico PDN-05-030, PlanetLab Consortium, junio de 2005. Citado en la página 99.

PFLEEGER, C.: *Security in Computing*. Upper Saddle River, NJ: Prentice Hall, 3a. ed., 2003. Citado en las páginas 378 y 394.

PICCO, G., BALZAROTTI, D. y COSTA, P.: "LighTS: A Lightweight, Customizable Tuple Space Supporting Context-Aware Applications." *Proc. Symp. Applied Computing* (Santa Fe, NM). Nueva York, NY: ACM Press, 2005, págs. 413-419. Citado en la página 595.

PIERRE, G. y VAN STEEN, M.: "Globule: A Collaborative Content Delivery Network." *IEEE Commun. Mag.* (44)8, agosto de 2006. Citado en las páginas 54 y 63.

PIERRE, G., VAN STEEN, M. y TANENBAUM, A.: "Dynamically Selecting Optimal Distribution Strategies for Web Documents." *IEEE Trans. Comp.* (51)6:637-651, junio de 2002. Citado en la página 64.

PIETZUCH, P. R. y BACON, J. M.: "Hermes: A Distributed Event-Based Middleware Architecture." *Proc. Workshop on Distributed Event-Based Systems* (Viena, Austria). Los Alamitos, CA: IEEE Computer Society Press, 2002. Citado en la página 633.

PIKE, R., PRESOTTO, D., DORWARD, S., FLANDRENA, B., THOMPSON, K., TRICKEY, H. y WINTERBOTTOM, P.: "Plan 9 from Bell Labs." *Computing Systems* (8)3:221-254, verano de 1995. Citado en las páginas 197 y 505.

PINZARI, G.: "NX X Protocol Compression." Reporte técnico D-309/3-NXP-DOC, NoMachine, Roma, Italia, septiembre de 2003. Citado en la página 84.

PITOURA, E. y SAMARAS, G.: "Locating Objects in Mobile Computing." *IEEE Trans. Know. Data Eng.* (13)4:571-592, julio de 2001. Citado en las páginas 192 y 627.

PLAINFOSSE, D. y SHAPIRO, M.: "A Survey of Distributed Garbage Collection Techniques." En *Proc. Int'l Workshop on Memory Management*, vol. 986 de *Lect. Notes Comp. Sc.*, págs. 211-249. Berlín: Springer-Verlag, septiembre de 1995. Citado en la página 186.

PLUMMER, D.: "Ethernet Address Resolution Protocol." RFC 826, noviembre de 1982. Citado en la página 183.

PODLING, S. y BOSZORMENYI, L.: "A Survey of Web Cache Replacement Strategies." *ACM Comput. Surv.* (35)4:374-398, diciembre de 2003. Citado en las páginas 573 y 632.

POPESCU, B., VAN STEEN, M. y TANENBAUM, A.: "A Security Architecture for Object-Based Distributed Systems." *Proc. 18th Ann. Computer Security Application Conf.* (Las Vegas, NA). ACSA, 2002. Citado en la página 482.

POSTEL, J.: "Simple Mail Transfer Protoco.l" RFC 821, agosto de 1982. Citado en la página 151.

POSTEL, J. y REYNOLDS, J.: "File Transfer Protocol." RFC 995, octubre de 1985. Citado en la página 122.

POTZL, H., ANDERSON, M. y STEINBRINK, B.: "Linux-VServer: Resource Efficient Context Isolation." *Free Software Magazine*, núm. 5, junio de 2005. Citado en la página 103.

POUWELSE, J., GARBACKI, P., EPEMA, D. y SIPS, H.: "A Measurement Study of the BitTorrent Peer-to-Peer File-Sharing System." Reporte técnico PDS-2004-003, Technical University Delft, abril de 2004. Citado en la página 53.

POUWELSE, J. A., GARBACKI, P., EPEMA, D. H. J. y SIPS, H. J.: "The Bittorrent P2P File-Sharing System: Measurements and Analysis." *Proc. Fourth Int'l Workshop on Peer-to-Peer Systems*, vol. 3640 de *Lect. Notes Comp. Sc.* (Itaca, NY). Berlín: Springer-Verlag, 2005, págs. 205-216. Citado en la página 527.

QIN, F., TUCEK, J., SUNDARESAN, J. y ZHOU, Y.: "Rx: Treating Bugs as Allergies -A Safe Method to Survive Software Failures." *Proc. 20th Symp. Operating System Principles* (Brighton, RU). Nueva York, NY: ACM Press, 2005, págs. 235-248. Citado en la página 372.

QIU, L., PADMANABHAN, V. y VOELKER, G.: "On the Placement of Web Server Replicas." *Proc. 20th INFOCOM Conf.* (Anchorage, AK). Los Alamitos, CA: IEEE Computer Society Press, 2001, págs. 1587-1596. Citado en las páginas 296 y 297.

RABINOVICH, M. y SPASTSCHECK, O.: *Web Caching and Replication*. Reading, MA: Addison-Wesley, 2002. Citado en las páginas 52, 570 y 633.

RABINOVICH, M., RABINOVICH, I., RAJARAMAN, R. y AGGARWAL, A.: "A Dynamic Object Replication and Migration Protocol for an Internet Hosting Service." *Proc. 19th Int'l Conf. on Distributed Computing Systems*. IEEE, 1999. págs. 101-113. Citado en la página 299.

RADIA, S.: *Names, Contexts, and Closure Mechanisms in Distributed Computing Environments*. Tesis de doctorado, University of Waterloo, Ontario, 1989. Citado en la página 198.

RADOSLAVOV, P., GOVINDAN, R. y ESTRIN, D.: "Topology-Informed Internet Replica Placement." *Proc. Sixth Web Caching Workshop* (Boston, MA). Amsterdam: Norte de Holanda, 2001. Citado en la página 296.

RAFAELI, S. y HUTCHISON, D.: "A Survey of Key Management for Secure Group Communication." *ACM Comput. Surv.* (35)3:309-329, septiembre de 2003. Citado en la página 631.

RAICIU, C. y ROSENBLUM, D.: "Enabling Confidentiality in Content-Based Publish/Subscribe Infrastructures." Reporte técnico RN/05/30, Department of Computer Science, University College, Londres, 2005. Citado en la página 619.

RAMANATHAN, P., SHIN, K. y BUTLER, R.: "Fault-Tolerant Clock Synchronization in Distributed Systems." *IEEE Computer* (23)10:33-42, octubre de 1990. Citado en la página 238.

RAMASUBRAMANIAN, V. y SIRER, E. G.: "The Design and Implementation of a Next Generation Name Service for the Internet." *Proc. SIGCOMM* (Portland, OR). Nueva York, NY: ACM Press, 2004a. Citado en la página 215.

RAMASUBRAMANIAN, V. y SIRER, E. G.: "Beehive: O(1) Lookup Performance for Power-Law Query Distributions in Peer-to-Peer Overlays." *Proc. First Symp. Networked Systems Design and Impl.* (San Francisco, CA). Berkeley, CA: USENIX, 2004b, págs. 99-112. Citado en las páginas 216 y 527.

RATNASAMY, S., FRANCIS, P., HANDLEY, M., KARP, R. y SCHENKER, S.: "A Scalable Content-Addressable Network." *Proc. SIGCOMM*. ACM, 2001, págs. 161-172. Citado en la página 45.

RAYNAL, M. y SINGHAL, M.: "Logical Time: Capturing Causality in Distributed Systems." *IEEE Computer* (29)2:49-56, febrero de 1996. Citado en las páginas 246 y 628.

REITER, M.: "How to Securely Replicate Services." *ACM Trans. Prog. Lang. Syst.* (16)3:986-1009, mayo de 1994. Citado en las páginas 409 y 411.

REITER, M., BIRMAN, K. y VAN RENESSE, R.: "A Security Architecture for Fault- Tolerant Systems." *ACM Trans. Comp. Syst.* (12)4:340-371, noviembre de 1994. Citado en la página 433.

RESCORLA, E. y SCHIFFMAN, A.: "The Secure HyperText Transfer Protocol." RFC 2660, agosto de 1999. Citado en la página 565.

REYNOLDS, J. y POSTEL, J.: "Assigned Numbers." RFC 1700, octubre de 1994. Citado en la página 89.

RICART, G. y AGRAWALA, A.: "An Optimal Algorithm for Mutual Exclusion in Computer Networks." *Commun. ACM* (24)1:9-17, enero de 1981. Citado en la página 255.

RISSON, J. y MOORS, T.: "Survey of Research towards Robust Peer-to-Peer Networks: Search Methods." *Comp. Netw.* (50), 2006. Citado en las páginas 47 y 226.

RIVEST, R.: "The MD5 Message Digest Algorithm." RFC 1321, abril de 1992. Citado en la página 395.

RIVEST, R., SHAMIR, A. y ADLEMAN, L.: "A Method for Obtaining Digital Signatures and Public-key Cryptosystems." *Commun. ACM* (21)2:120-126, febrero de 1978. Citado en la página 394.

RIZZO, L.: "Effective Erasure Codes for Reliable Computer Communication Protocols." *ACM Comp. Commun. Rev.*, (27)2:24-36, abril de 1997. Citado en la página 364.

RODRIGUES, L., FONSECA, H. y VERISSIMO, P.: "Totally Ordered Multicast in Large-Scale Systems." *Proc. 16th Int'l Conf. on Distributed Computing Systems*. IEEE, 1996, págs. 503-510. Citado en la página 311.

RODRIGUES, R. y LISKOV, B.: "High Availability in DHTs: Erasure Coding vs. Replication." *Proc. Fourth Int'l Workshop on Peer-to-Peer Systems* (Itaca, NY), 2005. Citado en la página 532.

RODRIGUEZ, P., SPANNER, C. y BIERSACK, E.: "Analysis of Web Caching Architecture: Hierarchical and Distributed Caching." *IEEE/ACM Trans. Netw.* (21)4:404-418, agosto de 2001. Citado en la página 571.

ROSENBLUM, M. y GARFINKEL, T.: "Virtual Machine Monitors: Current Technology and Future Trends." *IEEE Computer* (38)5:39-47, mayo de 2005. Citado en la página 82.

ROUSSOS, G., MARSH, A. J. y MAGLAVERA, S.: "Enabling Pervasive Computing with Smart Phones." *IEEE Pervasive Comput.* (4)2:20-26, abril de 2005. Citado en la página 25.

ROWSTRON, A.: "Run-time Systems for Coordination." En A. Omicini, F. Zambonelli, M. Klusch, y R. Tolksdorf (eds.), *Coordination of Internet Agents: Models, Technologies and Applications*, págs. 78-96. Berlín: Springer-Verlag, 2001. Citado en la página 607.

ROWSTRON, A. y DRUSCHEL, P.: "Pastry: Scalable, Distributed Object Location and Routing for Large-Scale Peer-to-Peer Systems." *Proc. Middleware 2001*, vol. 2218 de *Lect. Notes Comp. Sc*. Berlín: Springer-Verlag, 2001, págs. 329-350. Citado en las páginas 167, 191 y 216.

ROWSTRON, A. y WRAY, S.: "A Run-Time System for WCL." En H. Bal, B. Belkhouche, y L. Cardelli (eds.), *Internet Programming Languages*, vol. 1686 de *Lect. Notes Comp. Sc.*, págs. 78-96. Berlín: Springer-Verlag, 1998. Citado en la página 610.

RUSSELLO, G., CHAUDRON, M. y VAN STEEN, M.: "Adapting Strategies for Distributing Data in Shared Data Space." *Proc. Int'l Symp. Distr. Objects & Appl. (DOA)*, vol. 3291 de *Lect. Notes Comp. Sc.* (Agia Napa, Chipre). Berlín: Springer-Verlag, 2004, págs. 1225-1242. Citado en las páginas 611 y 612.

RUSSELLO, G., CHAUDRON, M., VAN STEEN, M. y BOKHAROUSS, I.: "Dynamically Adapting Tuple Replication for Managing Availability in a Shared Data Space." *Sc. Comp. Programming* (63), 2006. Citado en las páginas 611 y 616.

SADJADI, S. y MCKINLEY, P.: "A Survey of Adaptive Middleware." Reporte técnico MSU-CSE-03-35, Michigan State University, Computer Science and Engineering, diciembre de 2003. Citado en la página 55.

SAITO, Y. y SHAPIRO, M.: "Optimistic Replication." *ACM Comput. Surv.* (37)1:42-81, marzo de 2005. Citado en la página 628.

SALTZER, J. y SCHROEDER, M.: "The Protection of Information in Computer Systems." *Proceedings of the IEEE* (63)9:1278-1308, septiembre de 1975. Citado en la página 416.

SALTZER, J.: "Naming and Binding Objects." En R. Bayer, R. Graham y G. Seegmuller (eds.), *Operating Systems: An Advanced Course*, vol. 60 de *Lect. Notes Comp. Sc.*, págs. 99-208. Berlín: Springer-Verlag, 1978. Citado en la página 627.

SALTZER, J., REED, D. y CLARK, D.: "End-to-End Arguments in System Design." *ACM Trans. Comp. Syst.* (2)4:277-288, noviembre de 1984. Citado en la página 252.

SANDHU, R. S., COYNE, E. J., FEINSTEIN, H. L. y YOUMAN, C. E.: "Role-Based Access Control Models". *IEEE Computer* (29)2:38-47, febrero de 1996. Citado en la página 417.

SAROIU, S., GUMMADI, P. K. y GRIBBLE, S. D.: "Measuring and Analyzing the Characteristics of Napster and Gnutella Hosts." *ACM Multimedia Syst.* (9)2:170-184, agosto de 2003. Citado en la página 53.

SATYANARAYANAN, M.: "The Evolution of Coda." *ACM Trans. Comp. Syst.*, (20)2:85-124, mayo de 2002. Citado en la página 632.

SATYANARAYANAN, M. y SIEGEL, E.: "Parallel Communication in a Large Distributed System." *IEEE Trans. Comp.* (39)3:328-348, marzo de 1990. Citado en la página 505.

SAXENA, P. y RAI, J.: "A Survey of Permission-based Distributed Mutual Exclusion Algorithms." *Computer Standards and Interfaces* (25)2:159-181, mayo de 2003. Citado en la página 252.

SCHMIDT, D., STAL, M., ROHNERT, H. y BUSCHMANN, F.: *Pattern-Oriented Software Architecture – Patterns for Concurrent and Networked Objects*. Nueva York: John Wiley, 2000. Citado en las páginas 55 y 625.

SCHNEIDER, F.: "Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial." *ACM Comput. Surv.* (22)4:299-320, diciembre de 1990. Citado en las páginas 248, 303 y 480.

SCHNEIER, B.: *Applied Cryptography*. Nueva York: John Wiley, 2a. ed., 1996. Citado en la páginas 391 y 411.

SCHNEIER, B.: *Secrets and Lies*. Nueva York: John Wiley, 2000. Citado en las páginas 391 y 630.

SCHULZRINNE, H.: "The tel URI for Telephone Numbers". RFC 3966, enero de 2005. Citado en la página 569.

SCHULZRINNE, H., CASNER, S., FREDERICK, R. y JACOBSON, V.: "RTP: A Transport Protocol for Real-Time Applications." RFC 3550, julio de 2003. Citado en la página 121.

SEBESTA, R.: *Programming the World Wide Web*. Reading, MA: Addison-Wesley, 3a. ed., 2006. Citado en las páginas 547 y 633.

SHAPIRO, M., DICKMAN, P. y PLAINFOSSE, D.: "SSP Chains: Robust, Distributed References Supporting Acyclic Garbage Collection." Reporte técnico 1799, INRIA, Rocquencourt, Francia, noviembre de 1992. Citado en la página 184.

SHAW, M. y CLEMENTS, P.: "A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems." *Proc. 21st Int'l Comp. Softw. & Appl. Conf.*, 1997, págs. 6-13. Citado en la página 34.

SHEPLER, S., CALLAGHAN, B., ROBINSON, D., THURLOW, R., BEAME, C., EISLER, M. y NOVECK, D.: "Network File System (NFS) Version 4 Protocol." RFC 3530, abril de 2003. Citado en las páginas 201 y 492.

SHETH, A. P. y LARSON, J. A.: "Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases." *ACM Comput. Surv.* (22)3:183-236, septiembre de 1990. Citado en la página 299.

SHOOMAN, M. L.: *Reliability of Computer Systems and Networks: Fault Tolerance, Analysis, and Design*. Nueva York: John Wiley, 2002. Citado en la página 322.

SILBERSCHATZ, A., GALVIN, P. y GAGNE, G.: *Operating System Concepts*. Nueva York: John Wiley, 7a. ed., 2005. Citado en las páginas 197 y 624.

SINGH, A., CASTRO, M., DRUSCHEL, P. y ROWSTRON, A.: "Defending Against Eclipse Attacks on Overlay Networks." *Proc. 11th SIGOPS European Workshop* (Leuven, Bélgica). Nueva York, NY: ACM Press, 2004, págs. 115-120. Citado en la página 539.

SINGH, A., NGAN, T.-W., DRUSCHEL, P. y WALLACH, D. S.: "Eclipse Attacks on Overlay Networks: Threats and Defenses." *Proc. 25th INFOCOM Conf.* (Barcelona, España). Los Alamitos, CA: IEEE Computer Society Press, 2006. Citado en la página 539.

SINGHAL, M. y SHIVARATRI, N.: *Advanced Concepts in Operating Systems: Distributed, Database, and Multiprocessor Operating Systems*. Nueva York: McGraw-Hill, 1994. Citado en la página 364.

SIVASUBRAMANIAN, S., PIERRE, G. y VAN STEEN, M.: "Replicating Web Applications On-Demand." *Proc. First Int'l Conf. Services Comput.* (Shanghai, China). Los Alamitos, CA: IEEE Computer Society Press, 2004a, págs. 227-236. Citado en la página 580.

SIVASUBRAMANIAN, S., PIERRE, G., VAN STEEN, M. y ALONSO, G.: "GlobeCBC: Content-blind Result Caching for Dynamic Web Applications." Reporte técnico, Vrije Universiteit, Department of Computer Science, enero de 2006. Citado en la página 582.

SIVASUBRAMANIAN, S., SZYMANIAK, M., PIERRE, G. y VAN STEEN, M.: "Replication for Web Hosting Systems." *ACM Comput. Surv.* (36)3:1-44, septiembre de 2004b. Citado en las páginas 299, 573 y 629.

SIVASUBRAMANIAN, S., ALONSO, G., PIERRE, G. y VAN STEEN, M.: "GlobeDB: Autonomic Data Replication for Web Applications." *Proc. 14th Int'l WWW Conf.* (Chiba, Japón). Nueva York, NY: ACM Press, 2005, págs. 33-42. Citado en la página 581.

SIVRIKAYA, F. y YENER, B.: "Time Synchronization in Sensor Networks: A Survey." *IEEE Network* (18)4:45-50, julio de 2004. Citado en la página 242.

SKEEN, D.: "Nonblocking Commit Protocols." *Proc. SIGMOD Int'l Conf. on Management Of Data*. ACM, 1981, págs. 133-142. Citado en la página 359.

SKEEN, D. y STONEBRAKER, M.: "A Formal Model of Crash Recovery in a Distributed System." *IEEE Trans. Softw. Eng.*, (SE-9)3:219-228, marzo de 1983. Citado en la página 361.

SMITH, J. y NAIR, R.: "The Architecture of Virtual Machines." *IEEE Computer* (38)5:32-38, mayo de 2005. Citado en las páginas 80 y 81.

SMITH, J. y NAIR, R.: *Virtual Machines: Versatile Platforms for Systems and Processes*. San Mateo, CA: Morgan Kaufman, 2005. Citado en la página 625.

SNIR, M., OTTO, S., HUSS-LEDERMAN, S., WALKER, D. y DONGARRA, J.: *MPI: The Complete Reference – The MPI Core*. Cambridge, MA: MIT Press, 1998. Citado en la página 145.

SPEAKMAN, T., CROWCROFT, J., GEMMELL, J., FARINACCI, D., LIN, S., LESHCHINER, D., LUBY, M., MONTGOMERY, T., RIZZO, L., TWEEDLY, A., BHASKAR, N., EDMONSTONE, R., SUMANASEKERA, R. y VICISANO, L.: "PGM Reliable Transport Protocol Specification." RFC 3208, diciembre de 2001. Citado en la página 614.

SPECHT, S. M. y LEE, R. B.: "Distributed Denial of Service: Taxonomies of Attacks, Tools, and Countermeasures." *Proc. Int'l Workshop on Security in Parallel and Distributed Systems* (San Francisco, CA), 2004, págs. 543-550. Citado en la página 427.

SPECTOR, A.: "Performing Remote Operations Efficiently on a Local Computer Network." *Commun. ACM* (25)4:246-260, abril de 1982. Citado en la página 339.

SRINIVASAN, R.: "RPC: Remote Procedure Call Protocol Specification Version 2." RFC 1831, agosto de 1995a. Citado en la página 502.

SRINIVASAN, R.: "XDR: External Data Representation Standard." RFC 1832, agosto de 1995b. Citado en la página 502.

SRIPANIDKULCHAI, K., MAGGS, B. y ZHANG, H.: "Efficient Content Location Using Interest-Based Locality in Peer-to-Peer Systems." *Proc. 22nd INFOCOM Conf.* (San Francisco, CA). Los Alamitos, CA: IEEE Computer Society Press, 2003. Citado en la página 225.

STEIN, L.: *Web Security, A Step-by-Step Reference Guide*. Reading, MA: Addison-Wesley, 1998. Citado en la página 432.

STEINDER, M. y SETHI, A.: "A Survey of Fault Localization Techniques in Computer Networks." *Sc. Comp. Programming*, (53)165-194, mayo de 2004. Citado en la página 372.

STEINER, J., NEUMAN, C. y SCHILLER, J.: "Kerberos: An Authentication Service for Open Network Systems." *Proc. Winter Techn. Conf*. USENIX, 1988, págs. 191-202. Citado en la página 411.

STEINMETZ, R.: "Human Perception of Jitter and Media Synchronization." *IEEE J. Selected Areas Commun.* (14)1:61-72, enero de 1996. Citado en la página 163.

STEINMETZ, R. y NAHRSTEDT, K.: *Multimedia Systems*. Berlín: Springer-Verlag, 2004. Citado en las páginas 93, 157, 160 y 626.

STEVENS, W.: *UNIX Network Programming – Networking APIs: Sockets and XTI*. Englewood Cliffs, NJ: Prentice Hall, 2a. ed., 1998. Citado en las páginas 76 y 142.

STEVENS, W.: *UNIX Network Programming – Interprocess Communication*. Englewood Cliffs, NJ: Prentice Hall, 2a. ed., 1999. Citado en las páginas 70 y 136.

STEVENS, W. y RAGO, S.: *Advanced Programming in the UNIX Environment*. Reading, MA: Addison-Wesley, 2a. ed., 2005. Citado en las páginas 72 y 626.

STOICA, I., MORRIS, R., LIBEN-NOWELL, D., KARGER, D. R., KAASHOEK, M. F., DABEK, F. y BALAKRISHNAN, H.: "Chord: A Scalable Peer-to-peer Lookup Protocol for Internet Applications." *IEEE/ACM Trans. Netw.* (11)1:17-32, febrero de 2003. Citado en las páginas 44 y 188.

STOJMENOVIC, I.: "Position-based Routing in Ad Hoc Networks." *IEEE Commun.Mag.* (40)7:128-134, julio de 2002. Citado en la página 261.

STRAUSS, J., KATABI, D. y KAASHOEK, F.: "A Measurement Study of Available Bandwidth Estimation Tools." *Proc. Third Internet Measurement Conf.* (Miami Beach, FL, EUA). Nueva York, NY: ACM Press, 2003, págs. 39-44. Citado en la página 575.

SUGERMAN, J., VENKITACHALAM, G. y LIM, B. H.: "Virtualizing I/O Devices on VMware. Workstation s Hosted Virtual Machine Monitor." *Proc. USENIX Ann. Techn. Conf.* (Boston, MA). Berkeley, CA: USENIX, 2001, págs. 1-14. Citado en la página 81.

SUN MICROSYSTEMS: *Java Message Service, Version 1.1*. Sun Microsystems, Mountain View, CA., abril de 2004a. Citado en las páginas 466 y 593.

SUN MICROSYSTEMS: *Java Remote Method Invocation Specification, JDK 1.5*. Sun Microsystems, Mountain View, CA., 2004b. Citado en la página 122.

SUN MICROSYSTEMS: *EJB 3.0 Simplified API*. Sun Microsystems, Mountain View, CA., agosto de 2005a. Citado en la página 447.

SUN MICROSYSTEMS: *Jini Technology Starter Kit, Version 2.1*, octubre de 2005b. Citado en las páginas 486 y 593.

SUNDARARAMAN, B., BUY, U. y KSHEMKALYANI, A. D.: "Clock Synchronization for Wireless Sensor Networks: A Survey." *Ad-Hoc Networks* (3)3:281-323, mayo de 2005. Citado en la página 242.

SZYMANIAK, M., PIERRE, G. y VAN STEEN, M.: "Scalable Cooperative Latency Estimation." *Proc. Tenth Int'l Conf. Parallel and Distributed Systems* (Newport Beach, CA). Los Alamitos, CA: IEEE Computer Society Press, 2004, págs. 367-376. Citado en la página 263.

SZYMANIAK, M., PIERRE, G. y VAN STEEN, M.: "A Single-Homed Ad hoc Distributed Server." Reporte técnico IR-CS-013, Vrije Universiteit, Department of Computer Science, marzo de 2005. Citado en la página 96.

SZYMANIAK, M., PIERRE, G. y VAN STEEN, M.: "Latency-driven replica placement." *IPSJ Digital Courier* (2), 2006. Citado en la página 297.

TAIANI, F., FABRE, J. C., y KILLIJIAN, M. O.: "A Multi-Level Meta-Object Protocol for Fault-Tolerance in Complex Architectures." *Proc. Int'l Conf. Dependable Systems and Networks* (Yokohama, Japón). Los Alamitos, CA: IEEE Computer Society Press, 2005, págs. 270-279. Citado en la página 474.

TAM, D., AZIMI, R. y JACOBSEN, H. A.: "Building Content-Based Publish/Subscribe Systems with Distributed Hash Tables." *Proc. First Int'l Workshop on Databases, Information Systems and Peer-to-Peer Computing*, vol. 2944 de *Lect. Notes Comp. Sc.* (Berlín, Alemania). Berlín: Springer-Verlag, 2003, págs. 138-152. Citado en la página 597.

TAN, S. W., WATERS, G. y CRAWFORD, J.: "A Survey and Performance Evaluation of Scalable Tree-based Application Layer Multicast Protocols." Reporte técnico 9-03, University of Kent, RU, julio de 2003. Citado en la página 169.

TANENBAUM, A.: *Computer Networks*. Upper Saddle River, NJ: Prentice Hall, 4a. ed., 2003. Citado en las páginas 117 y 336.

TANENBAUM, A., MULLENDER, S. y VAN RENESSE, R.: "Using Sparse Capabilities in a Distributed Operating System." *Proc. Sixth Int'l Conf. on Distributed Computing Systems*. IEEE, 1986, págs. 558-563. Citado en la página 435.

TANENBAUM, A., VAN RENESSE, R., VAN STAVEREN, H., SHARP, G., MULLENDER, S., JANSEN, J. y VAN ROSSUM, G.: "Experiences with the Amoeba Distributed Operating System." *Commun. ACM* (33)12:46-63, diciembre de 1990. Citado en la página 415.

TANENBAUM, A. y WOODHULL, A.: *Operating Systems, Design and Implementation*. Englewood Cliffs, NJ: Prentice Hall, 3a. ed., 2006. Citado en las páginas 197 y 495.

TANISCH, P.: "Atomic Commit in Concurrent Computing." *IEEE Concurrency* (8)4:34- 41, octubre de 2000. Citado en la página 355.

TARTALJA, I. y MILUTINOVIC, V.: "Classifying Software-Based Cache Coherence Solutions". *IEEE Softw.* (14)3:90-101, mayo de 1997. Citado en la página 313.

TEL, G.: *Introduction to Distributed Algorithms*. Cambridge, RU: Cambridge University Press, 2a. ed., 2000. Citado en las páginas 232, 263 y 628.

TERRY, D., DEMERS, A., PETERSEN, K., SPREITZER, M., THEIMER, M. y WELSH, B.: "Session Guarantees for Weakly Consistent Replicated Data." *Proc. Third Int'l Conf. on Parallel and Distributed Information Systems* (Austin, TX). Los Alamitos, CA: IEEE Computer Society Press, 1994, págs. 140-149. Citado en las páginas 290, 293 y 295.

TERRY, D., PETERSEN, K., SPREITZER, M. y THEIMER, M.: "The Case for Nontransparent Replication: Examples from Bayou." *IEEE Data Engineering* (21)4:12-20, diciembre de 1998. Citado en la página 290.

THOMAS, R.: "A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases". *ACM Trans. Database Syst.* (4)2:180-209, junio de 1979. Citado en la página 311.

TIBCO: *TIB/Rendezvous Concepts, Release 7.4*. TIBCO Software Inc., Palo Alto, CA, julio de 2005. Citado en las páginas 54 y 595.

TOLIA, N., HARKES, J., KOZUCH, M. y SATYANARAYAN, M.: "Integrating Portable and Distributed Storage." *Proc. Third USENIX Conf. File and Storage Techn.* (Boston, MA). Berkeley, CA: USENIX, 2004. Citado en la página 523.

TOLKSDORF, R. y ROWSTRON, A.: "Evaluating Fault Tolerance Methods for Largescale Linda-like systems." *Proc. Int'l Conf. on Parallel and Distributed Processing Techniques and Applications*, vol. 2, (Las Vegas, NV), 2000, págs. 793-800. Citado en la página 616.

TOWSLEY, D., KUROSE, J. y PINGALI, S.: "A Comparison of Sender-Initiated and Receiver-Initiated Reliable Multicast Protocols." *IEEE J. Selected Areas Commun.* (15)3:398-407, abril de 1997. Citado en la página 345.

TRIPATHI, A., KARNIK, N., VORA, M., AHMED, T. y SINGH, R.: "Mobile Agent Programming in Ajanta." *Proc. 19th Int'l Conf. on Distributed Computing Systems*. IEEE, 1999, págs. 190-197. Citado en la página 422.

TUREK, J. y SHASHA, S.: "The Many Faces of Consensus in Distributed Systems." *IEEE Computer* (25)6:8-17, junio de 1992. Citado en las páginas 332 y 335.

UMAR, A.: *Object-Oriented Client/Server Internet Environments*. Upper Saddle River, NJ: Prentice Hall, 1997. Citado en la página 41.

UPnP Forum: "UPnP Device Architecture Version 1.0.1, diciembre de 2003. Citado en la página 26.

VAN RENESSE, R., BIRMAN, K. y VOGELS, W.: "Astrolabe: A Robust and Scalable Technology for Distributed System Monitoring, Management, and Data Mining." *ACM Trans. Comp. Syst.* (21)2:164-206, mayo de 2003. Citado en la página 61.

VAN STEEN, M., HAUCK, F., HOMBURG, P. y TANENBAUM, A.: "Locating Objects in Wide-Area Systems." *IEEE Commun.* (36)1:104-109, enero de 1998. Citado en la página 192.

VASUDEVAN, S., KUROSE, J. F. y TOWSLEY, D. F.: "Design and Analysis of a Leader Election Algorithm for Mobile Ad Hoc Networks." *Proc. 12th Int'l Conf. on Network Protocols* (Berlín, Alemania). Los Alamitos, CA: IEEE Computer Society Press, 2004, págs. 350-360. Citado en las páginas 267 y 268.

VEIGA, L. y FERREIRA, P.: "Asynchronous Complete Distributed Garbage Collection." *Proc. 19th Int'l Parallel & Distributed Processing Symp.* (Denver, CO). Los Alamitos, CA: IEEE Computer Society Press, 2005. Citado en la página 186.

VELÁZQUEZ, M.: "A Survey of Distributed Mutual Exclusion Algorithms." Reporte técnico CS-93-116, University of Colorado at Boulder, septiembre de 1993. Citado en la página 252.

VERISSIMO, P. y RODRIGUES, L.: *Distributed Systems for Systems Architects*. Dordrecht, Países Bajos: Kluwer Academic Publishers, 2001. Citado en la página 624.

VETTER, R., SPELL, C. y WARD, C.: "Mosaic and the World-Wide Web." *IEEE Computer* (27)10:49-57, octubre de 1994. Citado en la página 545.

VITEK, J., BRYCE, C. y ORIOL, M.: "Coordinating Processes with Secure Spaces." *Sc. Comp. Programming* (46)1-2, 2003. Citado en la página 620.

VOGELS, W.: "Tracking Service Availability in Long Running Business Activities." *Proc. First Int'l Conf. Service Oriented Comput.*, vol. 2910 de *Lect. Notes Comp. Sc.* (Trento, Italia). Berlín: Springer-Verlag, 2003. págs, 395-408. Citado en la página 336.

VOULGARIS, S. y VAN STEEN, M.: "Epidemic-style Management of Semantic Overlays for Content-Based Searching." *Proc. 11th Int'l Conf. Parallel and Distributed Computing (Euro-Par)*, vol. 3648 de *Lect. Notes Comp. Sc.* (Lisboa, Portugal). Berlín: Springer-Verlag, 2005, págs. 1143-1152. Citado en la página 226.

VOULGARIS, S., RIVI`ERE, E., KERMARREC, A. M. y VAN STEEN, M.: "Sub-2-Sub: Self-Organizing Content-Based Publish and Subscribe for Dynamic and Large Scale Collaborative Networks." *Proc. Fifth Int'l Workshop on Peer-to-Peer Systems* (Santa Bárbara, CA), 2006. Citado en la página 597.

VOYDOCK, V. y KENT, S.: "Security Mechanisms in High-Level Network Protocols." *ACM Comput. Surv.* (15)2:135-171, junio de 1983. Citado en la página 397.

WAH, B. W., SU, X. y LIN, D.: "A Survey of Error-Concealment Schemes for Real-Time Audio y Video Transmissions over the Internet." *Proc. Int'l Symp. Multimedia Softw. Eng.* (Taipei, Taiwán). Los Alamitos, CA: IEEE Computer Society Press, 2000, págs. 17-24. Citado en la página 162.

WAHBE, R., LUCCO, S., ANDERSON, T. y GRAHAM, S.: "Efficient Software-based Fault Isolation." *Proc. 14th Symp. Operating System Principles*. ACM, 1993, págs. 203-216. Citado en la página 422.

WALDO, J.: "Remote Procedure Calls y Java Remote Method Invocation." *IEEE Concurrency* (6)3:5-7, julio de 1998. Citado en la página 463.

WALFISH, M., BALAKRISHNAN, H. y SHENKER, S.: "Untangling the Web from DNS." *Proc. First Symp. Networked Systems Design y Impl.* (San Francisco, CA). Berkeley, CA: USENIX, 2004, págs. 225-238. Citado en la página 215.

WALLACH, D.: "A Survey of Peer-to-Peer Security Issues." *Proc. Int'l Symp. Softw. Security*, vol. 2609 de *Lect. Notes Comp. Sc.* (Tokio, Japón). Berlín: Springer-Verlag, 2002, págs. 42-57. Citado en la página 539.

WALLACH, D., BALFANZ, D., DEAN, D. y FELTEN, E.: "Extensible Security Architectures for Java." *Proc. 16th Symp. Operating System Principles*. ACM, 1997, págs. 116-128. Citado en las páginas 424 y 426.

WANG, C., CARZANIGA, A., EVANS, D. y WOLF, A. L.: "Security Issues and Requirements for Internet-Scale Publish-Subscribe Systems." *Proc. 35th Hawaii Int'l Conf. System Sciences*, vol. 9. IEEE, 2002, págs. 303-310. Citado en la página 618.

WANG, H., LO, M. K. y WANG, C.: "Consumer Privacy Concerns about Internet Marketing." *Commun. ACM* (41)3:63-70, marzo de 1998. Citado en la página 4.

WATTS, D. J.: *Small Worlds, The Dynamics of Networks between Order and Randomness*. Princeton, NJ: Princeton University Press, 1999. Citado en la página 226.

WELLS, G., CHALMERS, A. y CLAYTON, P.: "Linda Implementations in Java for Concurrent Systems." *Conc. & Comput.: Prac. Exp.* (16)10:1005-1022, agosto de 2004. Citado en la página 633.

WESSELS, D.: *Squid: The Definitive Guide*. Sebastopol, CA: O'Reilly & Associates, 2004. Citado en las páginas 556 y 572.

WHITE, S. R., HANSON, J. E., WHALLEY, I., CHESS, D. M. y KEPHART, J. O.: "An Architectural Approach to Autonomic Computing." *Proc. First Int'l Conf. Autonomic Comput.* (Nueva York, NY). Los Alamitos, CA: IEEE Computer Society Press, 2004, págs. 2-9. Citado en la página 625.

WIERINGA, R. y DE JONGE, W.: "Object Identifiers, Keys, and Surrogates–Object Identifiers Revisited." *Theory and Practice of Object Systems* (1)2:101-114, 1995. Citado en la página 181.

WIESMANN, M., PEDONE, F., SCHIPER, A., KEMME, B. y ALONSO, G.: "Understanding Replication in Databases y Distributed Systems." *Proc. 20th Int'l Conf. on Distributed Computing Systems*. IEEE, 2000, págs. 264-274. Citado en la página 276.

WOLLRATH, A., RIGGS, R. y WALDO, J.: "A Distributed Object Model for the Java System." *Computing Systems* (9)4:265-290, otoño de 1996. Citado en las páginas 460 y 472.

WOLMAN, A., VOELKER, G., SHARMA, N., CARDWELL, N., KARLIN, A. y LEVY, H.: "On the Scale and Performance of Cooperative Web Proxy Caching." *Proc. 17th Symp. Operating System Principles*. ACM, 1999, págs. 16-31. Citado en la página 571.

WU, D., HOU, Y., ZHU, W., ZHANG, Y. y PEHA, J.: "Streaming Video over the Internet: Approaches and Directions." *IEEE Trans. Circuits & Syst. Video Techn.* (11)1:1-20, febrero de 2001. Citado en la página 159.

YANG, B. y GARCIA-MOLINA, H.: "Designing a Super-Peer Network." *Proc. 19th Int'l Conf. Data Engineering* (Bangalore, India). Los Alamitos, CA: IEEE Computer Society Press, 2003, págs. 49-60. Citado en la página 51.

YANG, M., ZHANG, Z., LI, X. y DAI Y.: "An Empirical Study of Free-Riding Behavior in the Maze P2P File-Sharing System." *Proc. Fourth Int'l Workshop on Peer-to-Peer Systems*, Lect. Notes Comp. Sc. (Itaca, NY). Berlín: Springer-Verlag, 2005. Citado en la página 53.

YELLIN, D.: "Competitive Algorithms for the Dynamic Selection of Component Implementations." *IBM Syst. J.* (42)1:85-97, enero de 2003. Citado en la página 58.

YU, H. y VAHDAT, A.: "Efficient Numerical Error Bounding for Replicated Network Services." En Abbadi, Amr El, Michael L. Brodie, Chakravarthy, Sharma, Dayal, Umeshwar, Kamel, Nabil, Schlageter, Gunter, y Kyu-Young Whang (eds.), *Proc.26th Int'l Conf. Very Large Data Bases*, (El Cairo, Egipto). San Mateo, CA: Morgan Kaufman, 2000, págs. 123-133. Citado en la página 306.

YU, H. y VAHDAT, A.: "Design and Evaluation of a Conit-Based Continuous Consistency Model for Replicated Services." *ACM Trans. Comp. Syst.* (20)3:239–282, 2002. Citado en las páginas 277, 279 y 575.

ZHANG, C. y JACOBSEN, H. A.: "Resolving Feature Convolution in Middleware Systems." *Proc. 19th OOPSLA*, (Vancouver, Canadá). Nueva York, NY: ACM Press, 2004, págs. 188-205. Citado en la página 58.

ZHAO, B., HUANG, L., STRIBLING, J., RHEA, S., JOSEPH, A. y KUBIATOWICZ, J.:"Tapestry: A Resilient Global-Scale Overlay for Service Deployment." *IEEE J. Selected Areas Commun.* (22)1:41-53, enero de 2004. Citado en la página 216.

ZHAO, F. y GUIBAS, L.: *Wireless Sensor Networks*. San Mateo, CA: Morgan Kaufman, 2004. Citado en las páginas 28 y 624.

ZHAO, Y., STURMAN, D. y BHOLA, S.: "Subscription Propagation in Highly-Available Publish/Subscribe Middleware." *Proc. Middleware 2004*, vol. 3231 de *Lect. Notes Comp. Sc.* (Toronto, Canadá). Berlín: Springer-Verlag, 2004, págs. 274-293. Citado en la página 633.

ZHU, Q., CHEN, Z., TAN, L., ZHOU, Y., KEETON, K. y WILKES, J.: "Hibernator: Helping Disk Arrays Sleep through the Winter." *Proc. 20th Symp. Operating System Prin.* (Brighton, RU). Nueva York, NY: ACM Press, 2005, págs. 177-190. Citado en la página 632.

ZHUANG, S. Q., GEELS, D., STOICA, I. y KATZ, R. H.: "On Failure Detection Algorithms in Overlay Networks." *Proc. 24th INFOCOM Conf.* (Miami, FL). Los Alamitos, CA: IEEE Computer Society Press, 2005. Citado en la página 335.

ZOGG, J. M.: "GPS Basics." Reporte técnico GPS-X-02007, UBlox, marzo de 2002. Citado en la página 236.

ZWICKY, E., COOPER, S., CHAPMAN, D. y RUSSELL, D.: *Building Internet Firewalls*. Sebastopol, CA: O'Reilly & Associates, 2a. ed., 2000. Citado en la página 418.

ÍNDICE

A

B

C

D

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z